TUM School of Engineering and Design
Technical University of Munich

Master's Thesis in Mechatronics and Robotics

Analysis, Design and Implementation of a Real-Time Memory Bandwidth Monitor and Regulation System for RK3588

Supervisor Prof. Dr. Marco Caccamo

Advisor Dr. Andrea Bastoni

Author Haozheng Huang

Date 22.06.2024 in Munich

Disclaimer

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Munich, 22.06.2024

(Haozheng Huang)

Acknowledgement

First, I would like to thank my supervisor, Prof. Dr. Marco Caccamo, for giving me the chance to write my Master's thesis at the Chair of Cyber-Physical Systems in Production Engineering and for providing all the resources and facilities I needed to complete this work. His fantastic lectures were what first drew me to this exciting research field.

I am also really grateful to my advisor, Dr. Andrea Bastoni, for accepting me for this challenging yet exciting topic. He has been dedicated to all the difficulties I have met and his insightful advice always shed light on the mysterious puzzles. His constructive comments and suggestions have greatly improved this work. Despite my initial challenges and frustrations, his enthusiasm and encouragement have continuously motivated me throughout this journey. Besides his academic guidance, I truly appreciate his genuine concern for my personal well-being and future goals. I really appreciate it that I have followed his guidance all the way.

I would also like to thank my co-advisor, Dr. Alexander Zuepke, for his invaluable guidance. His warm heart and dedication have inspired me and helped me through various challenges. His expertise has significantly improved the quality of this thesis.

Additionally, I am grateful to the members of the lab, especially Yi Jiang and Zhihang Wei, who taught me many important details and provided practical assistance throughout my thesis. Their support has been crucial in my learning process.

Finally, I would like to acknowledge the support and understanding of my family and friends, who encouraged me when I was depressed, and everyone who has helped me in any aspect during this project.

Abstract

In modern multi-core platforms, the DRAM system is one of the most significant interference sources, and makes the system less predictable. This thesis focuses on reducing the interference in the DRAM system, including DRAM, DRAM controllers, and DRAM bus on the RK3588 platform, which is one of the first platforms that integrate the DynamIQ Shared Unit from Arm. After analysing the state-of-the-art memory bandwidth regulation system (MemGuard), the Jailhouse hypervisor is ported to the RK3588 platforms, which is one of the first platforms that integrate the DynamIQ Shared Unit from Arm. Then this thesis designs and implements a memory bandwidth monitor and regulation system, considering the specification of the platform, based on the same logic as the version of MemGuard on ZCU102. Experiments show that the memory bandwidth regulation system is able to regulate per-core memory bandwidth and provide performance isolation with smaller overhead than the previous platform ZCU102.

Contents

Chapter 1

Introduction

1.1 Background

Real-time systems are computing systems that must react within precise time constraints to events in the environment [1]. As a consequence, the correct behavior of these systems depends not only on the value of the computation, but also on the time at which the results are produced [2]. Real-time systems can be classified into hard real-time systems and soft real-time systems. In hard real-time systems, system failure or catastrophic consequences may happen if the system is unable to respond correctly within the expected time interval. Typical hard real-time systems are aeronautics, satellite control, automotive, etc [3]. In soft real-time systems, although a strict response time is not compulsory, it is generally better to meet the timing requirement as much as possible [4]. Examples for soft real-time systems are video players, computer games, etc.

A real-time system usually executes multiple tasks, and typically, those tasks have deadlines, which are values of physical time by which the task must be completed [4]. To conclude whether every task can complete within its deadline, it is necessary to conduct a scheduling analysis. As a prerequisite for scheduling analysis, it is required to obtain the Worst-Case Execution Time (WCET) as accurately as possible [5] [6].

Classical real-time systems have very conservative architectures, and use single-core CPUs and dedicated specialized devices to fulfill the requirements [3]. In the past decades, however, multi-core platforms have been increasingly adopted. They increase the performance of embedded systems and can integrate mixed-criticality application workloads into one single platform [7]. Nevertheless, adopting multi-core systems is also a challenge in terms of real-time capability. On multi-core platforms, applications are executed in parallel by different cores and they may simultaneously access shared resources. On the one hand, a wide range of shared resources typically can only serve a certain number of requests at a time, such as the DRAM controller and shared buses. When one task is accessing a resource, while another task attempts to access the same resource simultaneously, the arbitration mechanisms of the resource may delay some of the requests which leads to uncertain execution time of the tasks. On the other hand, some shared resources, e.g. shared last level caches, may also change their state due to one application, which causes another application using that resource to suffer

from a slowdown [8]. Due to these types of interference among tasks, the execution time becomes variable, which makes the predictability of response time and WCET analysis more complex [9].

In multi-core systems, sources of interference include shared caches, shared interconnects, shared DRAM, DRAM controllers, and logical units pipelines [10] [11] [12] [3]. Among these various interference sources, the DRAM system is considered as one of the most significant sources of interference [3] [12].

When the memory system is shared by different cores, the processing time of a memory request is highly variable as it depends on the location of the access and the state of DRAM chips and the DRAM controller [13]. This characteristic is further described at the beginning of Chapter 2. This is critical for real-time systems because there will be no guarantees for the task deadlines if the memory access time is not predictable.

This thesis focuses on reducing the interference caused by DRAM, DRAM buses, and DRAM controllers. This thesis analyses and designs a real-time memory bandwidth monitor and regulation system based on the Performance Monitor Unit (PMU) [14] [15] for the the state-of-the-art platform Rockchip RK3588 [16] [17] (Chapter 3). Regarding implementation, first, the hypervisor Jailhouse [18] is ported to the RK3588 (Section 4.1), and then the memory bandwith monitoring and regulation solution is implemented within the hypervisor (Section 4.2). The solution is therefore transparent to the operating system. Extensive testing of the solution will be presented and discussed (Chapter 5).

1.2 Content of Thesis

In Chapter 2, DRAM structure and causes of unpredictable DRAM access latency are discussed. They are followed by related work in this field, on which this thesis will be based. Then Chapter 3 analyses the RK3588 platform and exploits the hardware module, Performance Monitor Unit (PMU) (Section 3.3.2), for the memory bandwidth regulator. This chapter also refers to existing designs, especially MemGuard [13], and intent to adapt MemGuard to a the RK3588 platform. Next, the implementation details of the regulator is explained in Chapter 4, including enabling the hypervisor and implementing the regulator. Chapter 5 presents experiments and evaluation of the memory bandwidth regulator. This includes preparation, functionality of the regulator, the aspect of performance isolation, and the overhead measurement. Through this the design is validated and some caveats are discovered. Finally, the thesis is concluded in Chapter 6, where limitation and future work are also discussed.

Chapter 2

Background and Related Work

2.1 DRAM Background

The shared memory system on multi-core platforms introduces difficulties to predict the execution time of different tasks on the cores, because the required processing time of a main memory access may vary significantly.

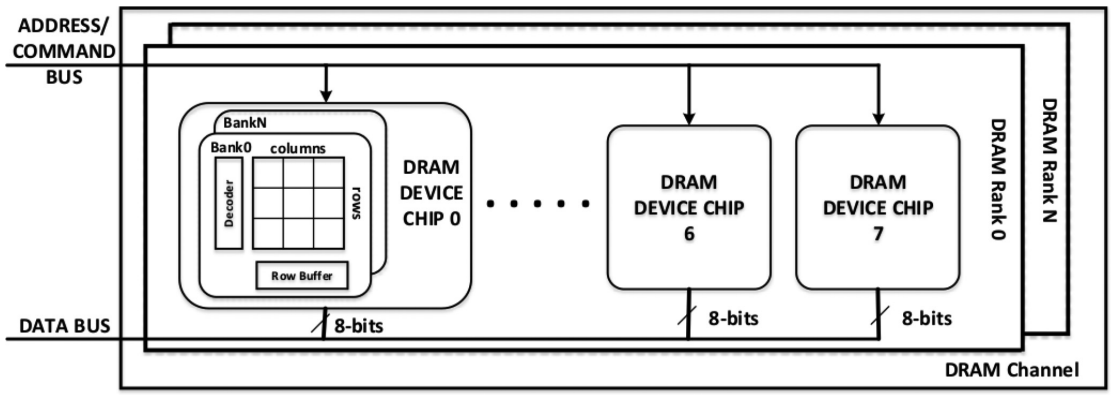


Figure 2.1: Architecture of memory controller and DRAM memory module [19].

As Figure 2.1 shows, DRAM has several chips, and each chip contains memory cells which are organized in banks, rows, and columns [19]. Banks can be operated in parallel, but all the banks on the same chip share the same data bus. Each bank has a row buffer which temporarily stores the copied data of the most recently accessed row. To retrieve the data in a specific cell, the row where the cell resides is activated (Activate command), and the whole row of data are stored into the corresponding row buffer. Then the column of that cell is selected in the row buffer (CAS command), and that specific subset of the row buffer is placed on the memory channel for data transmission [20].

Now if the memory controller wants to access another block of data in the same row, referred to as a row buffer hit, it simply issues another CAS command [20]. In contrast, the access latency is higher if the next data to be retrieved are in a different row in the same DRAM bank (row buffer miss), because the memory system has to additionally issue a Precharge command to write back the row buffer, and activate the target row [20]. Latency is

highest when every consecutive request targets a different row than the previous one. Hence, the concept of maximum sustainable bandwidth is introduced, which refers to the maximum bandwidth a memory controller can sustain under worst-case memory workload [21].

Moreover, the DRAM controller typically uses scheduling algorithms to re-order requests in order to maximize overall DRAM throughput [22], e.g. First Ready First-Come First-Serve (FR-FCFS) policy [23], and the scheduling algorithm implemented in the memory controller will influence the response time of memory requests [10]. To address the non-determinism derived from that, different techniques have been proposed.

2.2 Predictable DRAM Controller

A number of works focus on designing predictable DRAM controllers to produce tight WCET bounds [3]. This method develops specific hardware architecture to real-time performance.

Kim et al. [24] designed a DRAM controller with separated critical and non-critical memory access groups. Based on that, the proposed memory controller prioritizes the critical memory requests by preempting the non-critical requests [24], as an approach to meet deadlines for critical tasks and improve performance for non-critical tasks.

Valsan et al. [25] proposed a DRAM controller design called MEDUSA, which is also aimed to provide high time predictability for real-time tasks and high average performance for non-real-time tasks. In their approach, the OS partially partitions DRAM banks into two groups: reserved banks and shared banks. The reserved banks are exclusive to each core to provide predictable timing while the shared banks are shared by all cores to efficiently utilize the resources. MEDUSA has two separate queues for read and write requests, and it prioritizes reads over writes. In processing read requests, MEDUSA employs a two-level scheduling algorithm that prioritizes the memory requests to the reserved banks in a Round Robin fashion to provide strong timing predictability. In processing write requests, MEDUSA largely relies on the FR-FCFS for high throughput but makes an immediate switch to read upon arrival of read requests to the reserved banks.

Another novel memory controller, DRAMbulism [26], was proposed by Mirosanlou et al. to address the trade-off between average-case performance and predictable worst-case bounds. The proposed memory controller exploits the advantage of pipelining to improve both the performance and predictability. To tackle the difficulty in worst-case analysis introduced by pipelining effects, the proposal implements a simple acceptance rule that allows dynamically adding requests to the pipeline without violating worst-case bounds. The proposed design can provide comparable bounds to one of the most predictable real-time controllers [27] while delivering average performance similar to the high performance real-time controller [28] [26].

These proposals are claimed to be able to ensure predictability and improve performance. However, predictable DRAM controller designs usually require hardware support or modification. Since there is an increasing number of real-time systems built with Commercial Off-The-Shelf (COTS) components, hardware-based solutions are not generally possible in

such systems [13].

2.3 Memory Bandwidth Regulation

As discussed earlier in Section 2.1, a DRAM system can provide at least the maximum sustainable bandwidth. Following this principle, software memory bandwidth regulators emphasize the guaranteed memory bandwidth assigned to each CPU core, without modifying the DRAM hardware details.

Yun et al. [13] [29] proposed a memory bandwidth reservation system, MemGuard. MemGuard uses hardware performance counters (PMC) to track the number of memory transactions on each core within each predefined time interval (i.e. bandwidth). The predefined time interval is termed regulation periods. Each core is assigned a guaranteed bandwidth budget, and the budgets summed over all the cores should not exceed the sustainable bandwidth.

Once the budget of a core is depleted, the core is suspended. If the budgets of all the cores are exhausted while the current regulation period has not expired, MemGuard has two options to deliver the best-effort bandwidth. It either resumes all the cores and allow them to compete for the shared bandwidth until next regulation period, or starts the next regulation period immediately. The formerly strategy maximizes throughput as it is effectively equivalent to temporarily disabling the MemGuard, while the latter effectively makes each core to utilize the best-effort bandwidth proportional to its reserved bandwidth [29]. It should be noted that the MemGuards proposed in [13] and [29] are suitable for soft real-time applications, as they redistribute the per-core reserved bandwidth budgets based on previous bandwidth usage of the cores. If the regulator mis-predicts the necessary bandwidth for a core, the core may not use all its bandwidth budget and may miss the task deadline.

There are also variants of MemGuard for hard real-time tasks, e.g. the MemGuards in [30] and [31]. In contrast to the MemGuards mentioned earlier which are implemented in the operating system, these variants are implemented in the Jailhouse hypervisor [18]. This allows incorporating other techniques to further reduce interference among cores, such as cache-coloring [31].

MemGuard is able to enforce per-core guaranteed memory bandwidth. Nevertheless, MemGuard is based on the PMC interrupt and timer interrupt, and those interrupts introduce overhead to the processing cores. The overhead is more significant if the regulation period is shorter, e.g. around 8% overheads for periods of around 100 μs according to [29]. Moreover, it is challenging to implement complex performance metric that involves multiple counters in MemGuard [21].

To overcome those issues, MemPol [21] is proposed. Instead of using interrupt-based regulation, MemPol adopts a low-overhead, polling-based design that enables microsecond-scale memory bandwidth regulation and monitoring [21]. A dedicated core, e.g. the real-time core or the power-efficient core, is configured to poll the relevant performance counters. Furthermore, it relies on debug interfaces to suspend and resume the cores [21]. Through polling, it is also possible to combine performance counters readings to obtain complex performance

metric.

2.4 DRAM Partitioning

In modern operating systems, DRAM is treated as a single resource and OS may map virtual addresses of applications to different banks. As a result, applications are likely to share DRAM banks. This behaviour causes interference at DRAM level, as each bank can only open a single row at a time, and if multiple applications are contending for the data at the same bank, the memory access latency becomes unpredictable.

Although a bank can only manipulate one of its rows at a time, DRAM banks can be operated in parallel, as mentioned in 2.1. PALLOC [32] leverages this characteristic to improve isolation at DRAM level. PALLOC is an OS-level memory allocator which exploits the page-based virtual memory system to allocate memory pages on specific DRAM banks [32], so private DRAM banks can be assigned to different cores on the platform respectively. PALLOC significantly improves real-time performance without modifying any hardware, yet it is still far from ideal performance isolation due to other sources of interference [32].

2.5 Combination of Techniques

Mancuso et al. [33] proposed a technology that combines cache management, MemGuard and PALLOC into a multi-core platform, the Single Core Equivalence (SCE) technology.

In terms of management of shared caches, the proposed approach leverages a mechanism called Colored Lockdown [34] to partition shared caches and prevent frequently used memory pages from eviction. In addition, memory bandwidth is regulated by MemGuard [13] as mentioned above. Furthermore, SCE uses PALLOC [32] to assign private DRAM banks to applications on different cores. As a result, SCE exports a set of equivalent single-core virtual machines from a platform. Each set is an equivalent of a single-core platform, and thus existing software, schedulability analysis methodologies can be reused [33].

With all the techniques together, the proposed approach not only reduces interference at DRAM systems, but also at shared caches. Hence, SCE provides a strong isolation among cores on the platform, while on the other hand, it can cause memory resources to be underutilized [3].

Chapter 3

Analysis and Design

Referring to previous work on MemGuard [13] [29] [31] and MemPol [21], this thesis aims to discover whether the conventional memory bandwidth regulator is still applicable to the state-of-the-art platform RK3588 (Figure 3.1).

Compared to previous platforms, e.g. ZCU102 [35] where MemGuard and MemPol were implemented [31] [21], the RK3588 differs in processors, cache hierarchy, and interconnects, among others. While ZCU102 (Figure 3.2) has only four identical Cortex-A53 cores [36], RK3588 has eight cores: four efficient Cortex-A55 cores [14] and four performance Cortex-A76 cores [15]. ZCU102 has two levels of caches, but RK3588 has three levels of caches, with per-core L1 and L2 caches in respective cores and the L3 in the DynamIQ Shared Unit (DSU) [37]. Furthermore, ZCU102 uses CCI-400 [38] as interconnect, and RK3588 uses a proprietary interconnect from Rockchip Co., Ltd. which integrates DRAM schedulers that reorder requests to maximize DDR overall bandwidth [39]. All these differences make it less straightforward to design and implement a state-of-the-art memory bandwidth regulator on RK3588.

This chapter will start with the structure of RK3588, then analyse MemGuard and design the regulator based on it.

3.1 The RK3588 Platform

The RK3588 is one of the first platforms that integrate the Arm DynamIQ Shared Unit (DSU) [37] and features a quad-core set of Cortex-A76 [15] processors and a quad-core set of Cortex-A55 [14] processors. These processors are the first processors that can be integrated with the DSU and use a new connection schema for memory. The RK3588 platform has per-core L1 caches, per-core L2 caches, and a shared L3 cache in the DSU. Previous series of processors and chips e.g. Cortex-A72 [40] and Cortex-A53 [36], have only per-core L1 caches and they share a common L2 cache, and use a different interconnect, e.g. the CCI-400 on the ZCU102. Those preceding series have been analysed in studies e.g. [41] [31] [21]. However, the DynamIQ cluster [37] has not yet been investigated in terms of memory bandwidth regulation. Therefore, the RK3588 is selected as the target platform to bridge this gap in research.

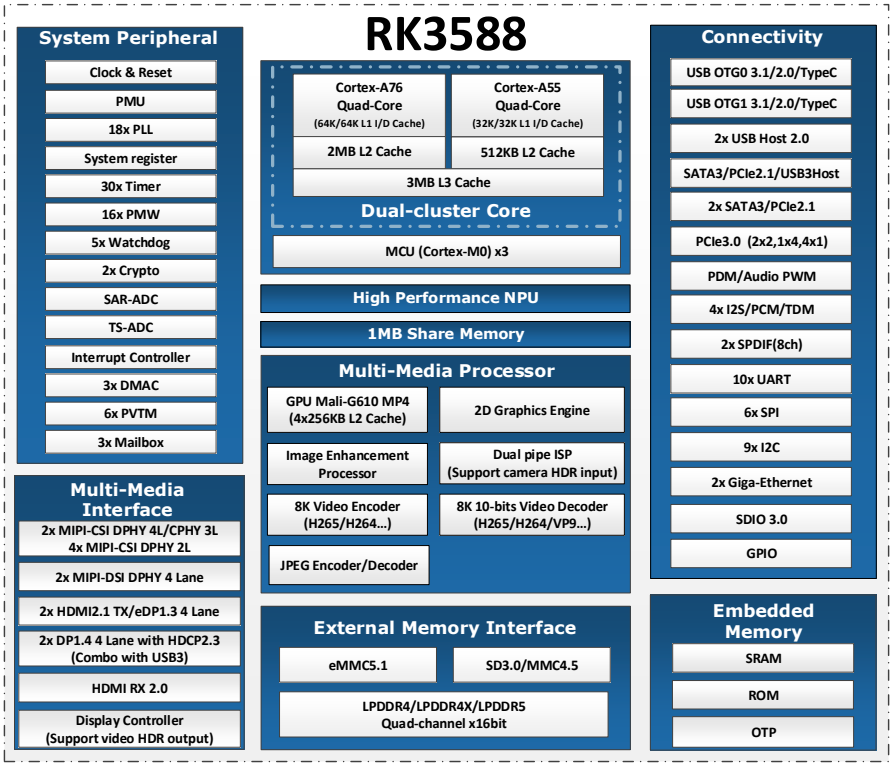


Figure 3.1: Block diagram of the RK3588 [17]. Each Cortex-A76 core as performance core has 64KB of L1 instruction cache, 64KB of L1 data cache, and 2MB of L2 cache; each Cortex-A55 core as efficient core has 32KB of L1 instruction cache, 32KB of L1 data cache, and 512KB of L2 cache. The two quad-core sets share the same 3MB of L3 cache of the DSU.

The Cortex-A76 and Cortex-A55 cores on the RK3588 implement the ARMv8-A architecture [14] [15] [42] with support for the v8.2 extension [39]. The Armv8-A architecture specifies four Exception Levels [43], which are basically the same concept as the privilege in modern computer systems. All the four Exception Levels are implemented on the A55 and A76 cores on the RK3588. The Exception Levels will be further explained in Section 3.2.1.

As Figure 3.3 shows, the RK3588 has three levels of caches, with L1 and L2 per-core caches and a L3 cache shared by the eight cores. The caches on both Cortex-A55 and Cortex-A76 adopt the write-back policy regarding writing data [14] [15]. The Cortex-A55 core and Cortex-A76 core differ in the cache inclusion policy, among others. For inclusive caches, a

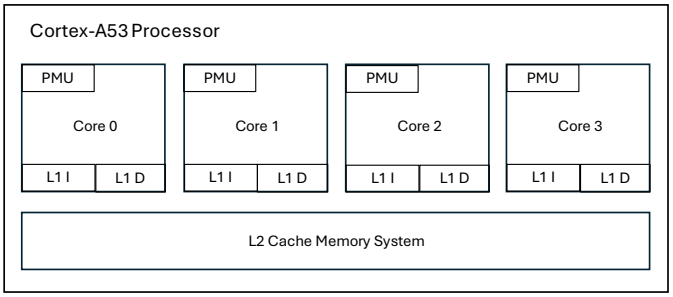


Figure 3.2: The processor and cache system on ZCU102.

cache line can reside in each level of the inclusive caches, while in contrast, exclusive caches only allow a cache line to reside in one cache level [44]. The L2 cache on the A55 core is strictly exclusive with L1 data cache [14], which means a cache line can only reside in either L1 or L2 cache. The L2 cache on the A76 core is strictly inclusive with L1 data cache [15], and therefore all the cache lines in L1 are also present in the L2. The L3 cache in the DSU, however, is exclusive if the cache line has only been used by one specific core since the cache line arrived in the cache hierarchy, but is inclusive when more than one core share that cache line [37]. These features require careful examination in terms of memory bandwidth regulation, and will be addressed in Section 3.3.4.

To monitor and regulate the memory bandwidth, certain hardware components are required as infrastructure. As shown in Figure 3.3, the processors and the DSU are the basic components of the cluster on the RK3588. Unlike the CCI-400 [38] on ZCU102 which can collect statistics of memory transactions on the interconnect, there is no dedicated hardware on the RK3588 that monitors the main memory bus directly. However, both the cores and the DSU have the Performance Monitoring Unit (PMU) (Section 3.3.2) which can provide implicit information about the number and the type of main memory accesses initiated by the cores. This is the hardware component used by MemGuard and MemPol to deduce the consumed memory bandwidth. Therefore, this thesis can rely on the PMU available on the RK3588 to monitor and regulate the memory bandwidth.

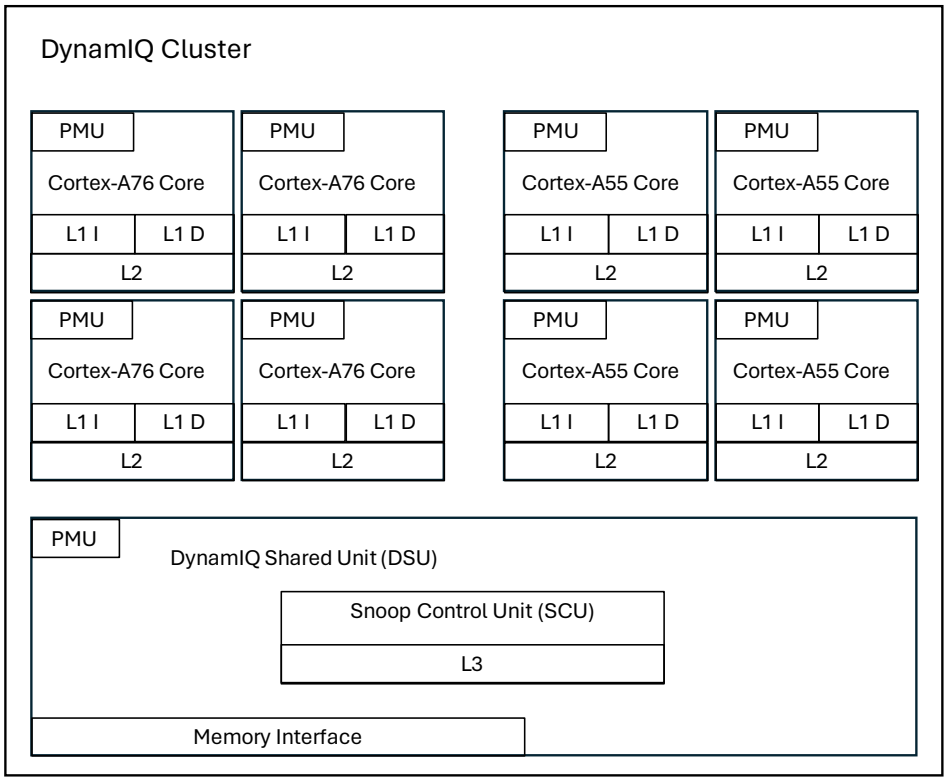


Figure 3.3: The DynamIQ Cluster on the RK3588 [37] [17]. The cluster includes a set of four Cortex-A76 cores, a set of four Cortex-A55 cores, and the DSU. It should be noted that not all the physical components are shown in the figure, but only those most relevant for the thesis.

3.2 Regulation within Hypervisor

Previous studies [45] [46] [31] [21] have integrated memory bandwidth regulation into a hypervisor. This is advantageous in terms of mixed-criticality tasks (Section 3.2.1). For simplicity and low overhead, the Jailhouse hypervisor [18] is commonly adopted, which is described in Section 3.2.2.

3.2.1 Exception Levels in the Arm Architecture

As mentioned in Section 3.1, there are four Exception Levels on the processors on the RK3588. The Exception Levels are numbered from 0 to 3, with 0 the weakest privilege and 3 the most powerful privilege, and they are normally abbreviated and referred to as EL0 to EL3 [43]. On the one hand, Exception Levels indicate the current execution of the processor, which means at which Exception Level the processor is executing the instructions. On the other hand, hardware resources and memory systems [43] are bonded with corresponding Exception Levels, so they are only accessible from the corresponding EL or higher Levels. For example, the register `MDCR_EL2` is only accessible when the processor is executing at EL2 or EL3. Typically, user applications execute at EL0, the operating system resides in EL1, the hypervisor operates at EL2, and the secure firmware is at EL3, although the Arm architecture does not specify which software uses which Exception Level [43].

The first MemGuards [13] [29] were designed and implemented in the Linux kernel for the x86 architecture. The implementation at operating system level allows memory bandwidth regulation with finer granularity with regards to different processes or threads. The regulator has the opportunity to better distribute the bandwidth by the leveraging detailed knowledge of application behavior and system states. In addition, all the hardware resources are already managed by the operating system, which reduces the effort to provide support for those resources.

However, the implementation at EL1 also has several disadvantages. First, the platform has to run only one single operating system that integrates the memory bandwidth regulation. Otherwise, if multiple operating systems or bare-metal programs are supposed to run in parallel on the platform, each OS has to integrate a regulator and they need a mechanism to exchange information of respective consumed bandwidth. Second, at EL1, the memory bandwidth regulator shares resources with other kernel functions and applications. This can lead to resource contention, where the regulator competes with other processes for CPU time and memory, potentially impacting its effectiveness.

Therefore, it would be beneficial to implement the regulation in a hypervisor, i.e. at EL2. In general, a hypervisor provides better isolation between cores. At EL2, the memory bandwidth regulator operates independently of any single operating system instance. This allows for multiple different operating systems or bare-metal programs to run on the platform, and thus the platform may execute mixed-criticality tasks. Nevertheless, the regulator itself is unable to run at EL2 as a standalone program. It needs to be integrated into a hypervisor

which manages hardware resources and isolation, e.g. stage 2 memory translation, for the virtual machines (VMs) across the platform.

3.2.2 The Jailhouse Hypervisor

Jailhouse is a partitioning hypervisor based on Linux, which is able to partition essential resources e.g. memory virtually and run bare-metal applications or operating systems in those partitions [18]. With the goal of optimal simplicity, Jailhouse is designed for static partitioning use cases [47], and does not support scheduling or any device emulation [48].

Jailhouse leverages the Linux kernel to boot and initialize itself, and once it is activated, it runs as a bare-metal program and takes full control over the hardware [18]. The hardware resources, e.g. CPU cores, RAM, or peripheral devices, are split into partitions which are called "cells" in Jailhouse. Each partition is a complete set of resources which are able to support OS or bare-metal programs. Figure 3.4 shows the overview when Jailhouse is running. The cell where Jailhouse is booted is called "root cell", and this cell also serves as an interface to manage Jailhouse. During activation of Jailhouse, the kernel module of Jailhouse loads Jailhouse into a reserved memory region. Then Jailhouse initializes CPUs and establish page tables, etc. Cells are only allowed to access the resources assigned to them.

Jailhouse is able to provide strong and clean isolation, delivering low latency and real-time performance [48]. Thanks to these advantages, Jailhouse is widely adopted and actively extended in research and industry. Mechanisms have been continuously integrated into Jailhouse to improve predictability of real-time systems [47], e.g. cache coloring in [46] [31], memory bandwidth regulation in [31] [30] [21], and device quality of service (QoS) in [30].

Yet, Jailhouse is currently not available for the RK3588, and it requires some modifications to the Linux operating system. This will be described more in details in Section 4.1.

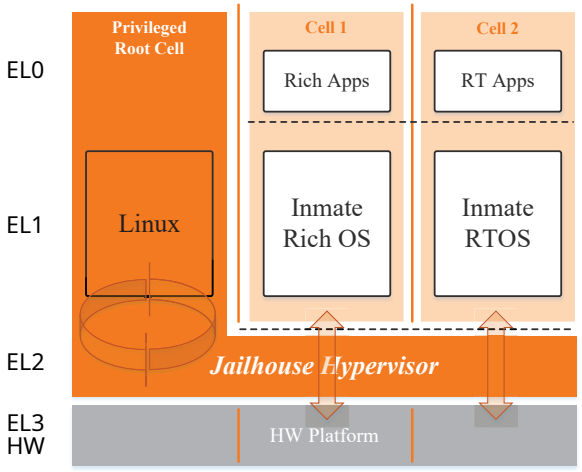


Figure 3.4: The architectural overview of Jailhouse [47].

3.3 The Memory Bandwidth Regulator - MemGuard

MemGuard is first proposed in [13]. The main idea is to estimate the consumed memory bandwidth by the Performance Monitor Counters (PMC), and stall or restore the execution of the processing core through a mechanism based on the Performance Monitor Unit (PMU). The PMU is a hardware component which can collect cache system statistics, and will be further explained in 3.3.2.

The regulator in this thesis refers to MemGuard and is adapted for the target platform RK3588.

3.3.1 Estimating Memory Bandwidth by Cache System Statistics

In general, MemGuard effectively uses cache system statistics to estimate the memory bandwidth. The statistics typically includes in each level the number of cache hits, misses, refills, write-backs, etc.

In computer systems, the processor queries the cache system in a hierarchical manner for the cache lines where the desired data is stored. The cache system always operates on the unit of a cache line, e.g. 64 Bytes. A cache hit at a certain level refers to the case when the processor finds the requested data in a cache line at that level of cache. If the data are not located at the level, it is termed a cache miss [49].

A cache miss prompts access to the lower level in the memory hierarchy to retrieve the cache line containing the requested data, which is a cache refill. In contrast to refill, a write-back occurs when the processor writes new value to the cache line. The cache line is then marked as "dirty", and later will be written to the lower level of the hierarchy when the cache system decides to evict it to accommodate new data [49].

From the cache system behaviour explained above, the consumed memory bandwidth can be estimated by the cache system statistics. For example, in the paper of the first MemGuard [13], the number of last-level cache (LLC) misses is considered as a metric for the estimation. It can be inferred that a cache miss at LLC triggers a refill loading data from outside of the core. Therefore, read requests of data are issued to the DRAM and cost memory bandwidth. The size of the transferred data corresponds to the size of a cache line.

If later the data is modified and needs to be written-back to the main memory, then there will be write accesses to the DRAM. Thus, the number of LLC misses should be doubled to reflect the consumed memory bandwidth in the worst-case scenario where a refill is always followed by a write-back [30]. However, this method has a drawback as mentioned in [21], that write-only workloads which do not cause cache line refills are not accounted for.

That is one of the methods to estimate the memory bandwidth used by the core. For the RK3588, a model is proposed in [50] and is adopted in this thesis. The model will be illustrated in Section 3.3.4.

3.3.2 Performance Monitor Unit (PMU)

In the Arm architecture, the Performance Monitor Unit (PMU) is designed to collect various statistics on the operation of the core and its memory system during runtime, which allows for detailed analysis of the core. It is also one of the profiling and debugging features on the Arm Cortex-A cores. The PMU has several PMC, and each can be individually configured to count the available events listed by the specification [14] [15]. The PMC, though, is a generic concept, and is not unique to the Arm architecture. It refers to a hardware component that tracks the number of specified system performance events, e.g. LLC miss in the paper [13].

The PMU registers can be accessed as system registers from the core [36], like MemGuard [13] [31], or from the external debug interface [36] based on the CoreSight infrastructure [51], like MemPol [21].

The PMU is also able to generate interrupts on PMC overflow, which provides the fundamental support for the regulation mechanism.

3.3.3 Regulation Mechanism

The mechanism behind the regulation is timer interrupt and PMU interrupt. Algorithm 1 shows the overview of the regulation mechanism. The timer is configured to generate an interrupt periodically. The period of the timer interrupt on each core is termed regulation period, which has a typical value of 1 ms [13] [21]. The PMU is configured to generate an interrupt upon the PMC overflow, and PMC counts the number of specified cache events.

As shown in Figure 3.5, at the beginning, the PMC on each core is initialized with a value so that it will overflow after the given number of cache events, i.e. when budget is depleted. The value is calculated by subtracting the budget from the maximum value, e.g. `0xffffffff` for a 32-bit counter. When the PMC overflows, the PMU interrupt handler triggers MemGuard regulation that stops the core and prevents it from executing tasks that consume bandwidth. The core is resumed when the next timer interrupt arrives. The timer interrupt handler resets the PMU value, which means assigning a new budget for the next regulation period to the core, and restores the core to run its given tasks if the core was blocked in last regulation period.

As mentioned in Chapter 2, the memory bandwidth varies greatly due to various reasons. To ensure predictability, the sum of assigned bandwidth to each core should be equal or less than the maximum sustainable bandwidth, which is the bandwidth the DRAM subsystem can provide in worst-case scenario.

In this way, the core never consumes more bandwidth than the assigned bandwidth, which is the PMC budget divided by the regulation period.

Algorithm 1: MemGuard Regulation Mechanism

```
1  init:
2      P_i ← regulation period for core i
3      Q_i ← budget for core i
4      register timer_interrupt_handler
5      register pmu_interrupt_handler

6  timer_interrupt_handler:
7      PMC_i ← max_value − Q_i
8      restore core i from possible blocking

9  pmu_interrupt_handler:
10     block core i from original execution
```

3.3.4 Adaptation for the RK3588

The principle of MemGuard could be applied on the RK3588, since it also has PMC which is a fundamental component of the regulation mechanism. Though, the regulator needs to be adapted to the platform. The processing cores and cache system on RK3588 are different from previous platforms e.g. ZCU102, as described at the beginning of Chapter 3. Therefore, the platform requires an appropriate model to estimate memory bandwidth, and the PMU should be adapted to that model. Since the Generic Interrupt Controller (GIC) [52] on RK3588 is a different version (GICv3) than the ones on older platforms, this should also be adapted. Implementation details will be described in Section 4.2.

Cache System of the RK3588

As described in Section 3.1, the RK3588 has three levels of cache. The statistics at L3 would be most interesting for estimating DRAM bandwidth, because it is the last level cache, and a cache miss or a cache write-back at this level implies transactions with DRAM. However, the L3 cache is shared by all the eight cores. Therefore, it should be checked whether the statistics can be attributed to the core to ensure per-core bandwidth estimation.

Furthermore, the adopted write-back policy should also be considered when designing the estimation model. For example, one write-back from L2 to L3 does not necessarily triggers one write (of the size of a cache line) to the main memory, but a write-through from L2 to L3 implies one write access to the main memory.

In addition, the estimation is also affected by the cache inclusion policy described in Section 3.1. An example for this could be the scenario where a PMC is configured to count how many cache lines the core has allocated into the private L2 cache. When the processor requests a cache line of data that is not found at L1 but found at L2, the cache line is moved from L2 into L1 for exclusive caches, or the cache line is copied into L1 for inclusive caches. If later the cache line at L1 needs to be evicted, for exclusive caches, the processor needs to allocate a cache line in the L2 cache to accommodate that cache line from L1. Consequently, the PMC count is incremented. However, for inclusive caches, no cache line needs to be

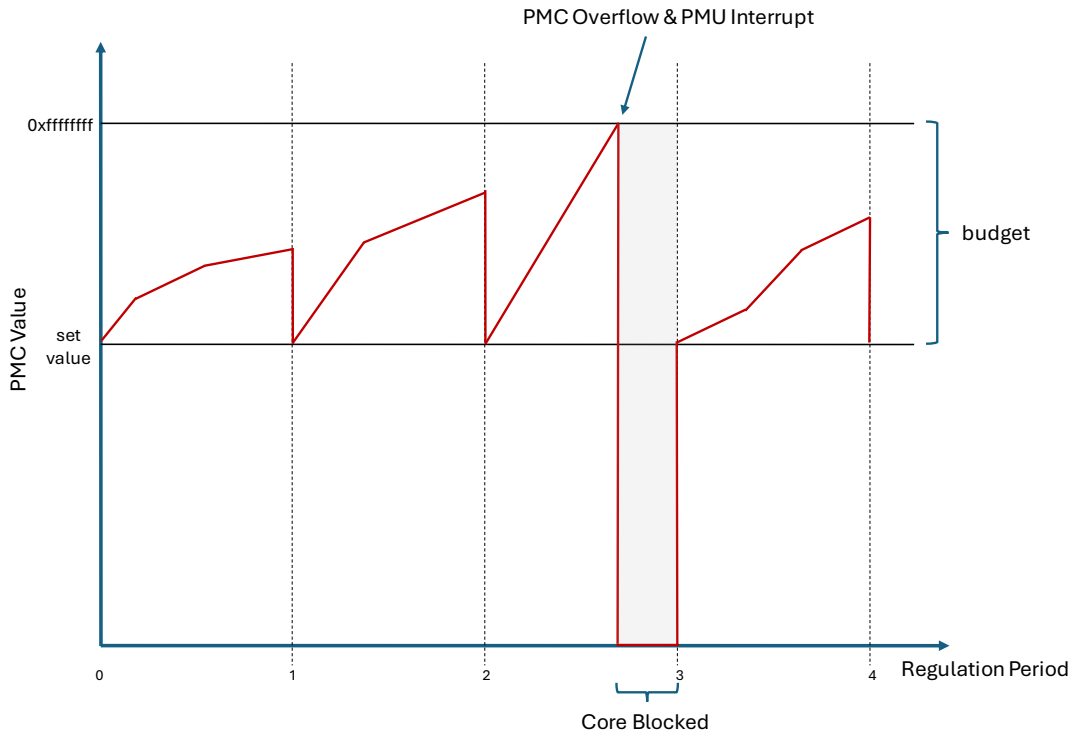


Figure 3.5: The regulation behaviour of MemGuard. The PMC is programmed to count from a set value. The difference between the maximum value of the counter and the set value is the budget assigned to the core. At the beginning of each regulation period, the PMC value is reset. If the PMC overflows, then the PMU interrupt is triggered and the interrupt handler blocks the core (gray area of the figure).

allocated in the L2, since the L2 cache has been holding the same copy of that cache line, and therefore the PMC is not incremented.

The next question is, which PMU event(s) should be chosen as the metric for the bandwidth estimation.

Memory Bandwidth Estimation Model

Each PMU on the A55 core and the A76 core has six PMC. For MemGuard, only one PMC on each PMU is used, since it is challenging to define complex regulation logic that combines more than a single type of PMU event on interrupt basis [21].

Referring to Jiang's Master's thesis [50], the PMU events `L3D_CACHE_ALLOCATE` for A55 cores and `L2D_CACHE_ALLOCATE` for A76 cores could be used to estimate the memory bandwidth respectively. Despite the descriptions of the PMU events in the technical reference manuals [14] [15], extensive experiments in [50] show some interesting results.

Before explaining those PMU events, some concepts need to be clarified. In this thesis, a cache line read means loading data of a cache line size into the cache. A standard write means a full cache line write into the cache, which does not cause a linefill, because this operation does not depend on the original data in that cache line. A write that only modifies

part of a cache line, e.g. only writing several bytes, is regarded as a modify. A modify requires the cache system to first load the data into a cache line, then part of the cache line is modified, and finally the cache line will be written-back to lower level of memory. Therefore, the size of transmitted data caused by a modify is the double of the one caused by a read or a standard write.

For the A55 core, statistically, the number of `L3D_CACHE_ALLOCATE` events corresponds to the sum of 1) cache line reads from outside the DynamIQ Cluster, 2) cache line writes to outside the DynamIQ Cluster, and 3) cache line modifies which involve data from outside of the DynamIQ Cluster, according to [50]. In other words, this event count reflects the sum of both directions of transactions (loading and storing data) with the interconnect (Figure 3.6).

For the A76 core, the event `L2D_CACHE_WR` has the similar pattern.

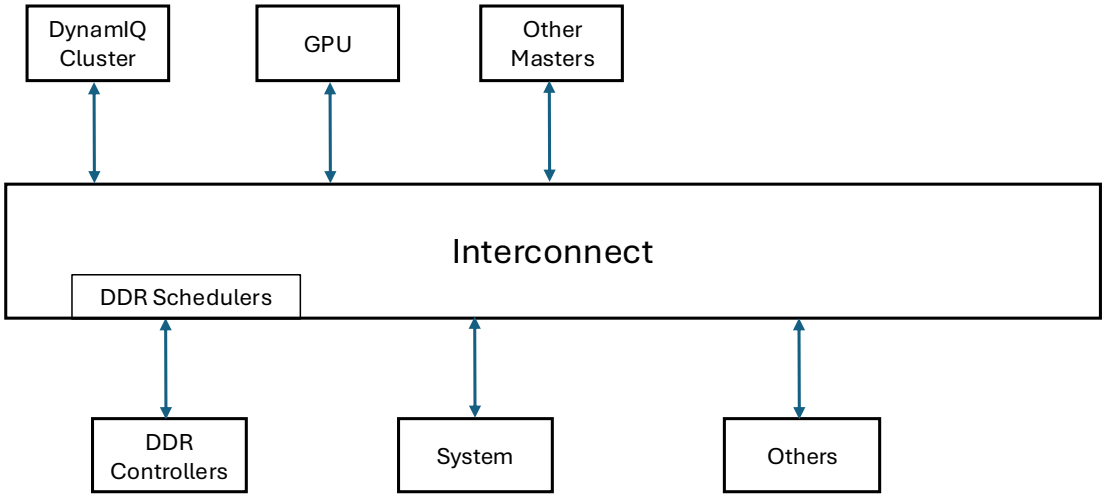


Figure 3.6: Overview of the interconnect on RK3588 [39]. The DynamIQ Cluster is the cluster of quad-Cortex-A55 cores, quad-Cortex-A76 cores, and the DSU, as described in Section 3.1.

In this regard, pessimistic models are proposed by [50] to estimate the size of transferred data by PMU events. The models are pessimistic as they assume every counted event corresponds to a cache line modify, and therefore could be always over-estimating the transferred data size. Based on those models, the memory bandwidth can be estimated as follows. For the A55 core,

$$consumed\_bandwidth_{A55}^{pess} = \frac{L3D\_CACHE\_ALLOCATE \cdot 2 \cdot cache\_line\_size}{regulation\_period}, \tag{3.1}$$

and for the A76 core,

$$consumed\_bandwidth_{A76}^{pess} = \frac{L2D\_CACHE\_WR \cdot 2 \cdot cache\_line\_size}{regulation\_period}. \tag{3.2}$$

If it is known in advance, e.g. through profiling, that the application only does full-line reads or full-line writes, then it is not the worst-case scenario. The above models can thus be optimized into the optimistic models: for the A55 core,

$$consumed\_bandwidth_{A55}^{opt} = \frac{L3D\_CACHE\_ALLOCATE \cdot cache\_line\_size}{regulation\_period}, \tag{3.3}$$

and for the A76 core,

$$consumed\_bandwidth_{A76}^{opt} = \frac{L2D\_CACHE\_WR \cdot cache\_line\_size}{regulation\_period}. \tag{3.4}$$

Chapter 4

Implementation

This chapter describes the implementation of the memory bandwidth regulator. First, the hypervisor is ported to the RK3588 platform which serves as the carrier of the regulator. Then the hardware components of the RK3588 are configured to ensure the regulator operates properly.

4.1 Porting Jailhouse to RK3588

As mentioned in Section 3.2.2, Jailhouse is a hypervisor that has been widely adopted in many works due to its lightweight and simplicity.

Since Jailhouse is started by a normal Linux system [18], to enable Jailhouse on the RK3588, some patches to both the Linux operating system [53] and Jailhouse are required. Furthermore, the address translation needs to be configured for the Linux operating system to access necessary hardware.

4.1.1 Patches to the Linux Kernel

The board used in the thesis is the Orange Pi 5 Plus (16GB) [54], and the manufacturer provides already a patched Linux kernel of the version 5.10 [55] which can run on the board [56]. Combined with the necessary patches from [53], the source code of the kernel is ready for compilation. Before the Linux kernel is compiled, some kernel configurations should be examined. `CONFIG_KVM` and `CONFIG_ARM64_VHE` (Virtualization Host Extension) should be disabled [57], and as discovered in the implementation process, the `CONFIG_ARM_GIC_V3_ITS` should be disabled, because it is the kernel support for the rather new feature Interrupt Translation Service (ITS) introduced by Arm [58] [52] and not yet supported by Jailhouse.

4.1.2 Patches to the Jailhouse

Regarding Jailhouse, some patches are necessary for the RK3588. The Linux kernel provided by the manufacturer [55] uses the System Control and Management Interface (SCMI) [59] to signal the firmware for system management, including power and performance functions. The SCMI specifies that the operating system at EL1 can choose to make Secure Monitor Calls (SMC) [43] [60] to notify the firmware. This is indeed implemented in the Linux kernel [55]. However, in Jailhouse [18], SMC calls are intercepted, and not all of the SMC calls are forwarded to the firmware. This would cause the CPU cores to be blocked when Jailhouse is activated, as they are waiting for firmware processing, but the firmware is not aware of the requests. Therefore, the patch [61] is needed to allow SMC calls to reach the firmware.

Apart from that, Jailhouse attempts to configure the system register `FPEXC32_EL2` when it tries to reset the CPU to create a separate partition of resources, or in other words, a cell [18] [62]. The register `FPEXC32_EL2` is floating-point exception control register [14] which is available on the Cortex-A55 core. However, the Cortex-A76 core does not generate floating-point exceptions [15], and therefore this register does not exist on the A76 core. To prevent the system from crashing on the A76 core, the corresponding code should be modified or discarded directly, since the floating-point exception is not a focus in the context of the thesis.

The firmware on RK3588 supports the Software Delegated Exception Interface (SDEI), which provides a mechanism for registering and servicing system events from system firmware [63]. Jailhouse chooses to use the SDEI for interrupt management if the platform advertises that it supports SDEI. However, for the RK3588, the timer interrupt and PMU interrupt are difficult to coordinate through the SDEI. Thus, Jailhouse is modified so that it does not use the SDEI for interrupt management on the RK3588.

All the developed extensions and changes have been committed in the repository[1] at the Chair of Cyber-Physical Systems in Production Engineering.

4.1.3 Configuring Jailhouse for RK3588

Jailhouse requires a piece of contiguous memory to place itself [18]. If there should be guest cells, then memory should also be reserved for them. For the Arm platform, the memory regions can be reserved in the device tree.

As Jailhouse serves as a hypervisor, it should translate the memory address that the operating system at EL1 requests into the physical address, which is known as stage 2 translation. To fulfill this, Jailhouse needs to know the address mapping in advance, especially the addresses that are mapped to memory-mapped devices. For the root cell, the addresses from Linux side map one-to-one to the physical addresses. The necessary addresses are those of

[1] https://gitlab.lrz.de/chair-of-cyber-physical-systems-in-production-engineering/thesis/haozheng_huang/jailhouse/-/tree/cps/orangepi5p-rk3588-haozheng-thesis

the serial console (UART), Generic Interrupt Controller (GIC) [52], and DRAM regions used by the Linux kernel. For simplicity, the thesis maps all the memory regions in Jailhouse.

For the bare-metal inmate, only the assigned memory regions and serial console are necessary. It should be noted that the inmate application believes that its memory address starts from `0x0`, which is the similar idea as the virtual address of applications running on the operation system, and therefore, Jailhouse should be configured to map the virtual address of the inmate to the physical address.

The root cell and the inmate can communicate with each other by shared memory, which is the same physical memory region configured in the Jailhouse address mapping.

4.2 Memory Bandwidth Regulator

Now that Jailhouse is ready on the RK3588, the memory bandwidth regulator can be implemented inside the hypervisor. A version of MemGuard already exists on the ZCU102 [31] [21]. This thesis keeps the same logic (explained in Section 3.3.3), and re-implements the regulator to fit the specific details of RK3588, i.e., the Generic Interrupt Controller (GIC) (Section 4.2.1), the PMU (Section 4.2.2), and the timer (Section 4.2.3).

4.2.1 Generic Interrupt Controller (GIC)

In MemGuard, a timer interrupt and a PMU interrupt are used in each core, so the interrupts needs to be configured. The Generic Interrupt Controller v3 (GICv3) on the RK3588 takes interrupts from peripherals, prioritizes them, and delivers them to the appropriate processor core [52] [39].

The Arm Generic Interrupt Controller architecture specifies several types of interrupts, and both the timer interrupt and PMU interrupt on the RK3588 are Private Peripheral Interrupts (PPIs) [64] [52] [39]. PPIs are peripheral interrupts private to one core and can only be handled by that core [52]. This simplifies the implementation to a certain extent compared to the ZCU102 [31] [21] which has the PMU interrupts as Shared Peripheral Interrupt (SPI) [35], because the SPI needs to be routed to a core to be handled [64].

The register interface of a GICv3 interrupt controller is split into three groups: Distributor interface, Redistributor interface, and CPU interface. The Distributor interface is used to configure SPIs and thus not relevant for the thesis. The CPU interface manages the acknowledgment, prioritization, routing, and masking of interrupts, which are mostly configured by Jailhouse or possibly the firmware already. The Redistributor is a per-core component that configures the PPIs that are used in the memory bandwidth regulator. It is necessary to enable or disable the corresponding interrupts through the Redistributor interface. It should be noted that, contrary to the SPIs used in the ZCU102, when Jailhouse is activated or instructed to create cells on the RK3588, it disables all the PPIs except the maintenance interrupt, and therefore the timer interrupt and PMU interrupt should be enabled again.

Other setup, e.g. routing controls and vector table, is already conducted by Jailhouse and does not require additional effort.

4.2.2 Bandwidth Reservation

To ensure the CPU core is blocked from original execution when the memory budget is depleted, it is required that the interrupt on PMC overflow is enabled, the interrupt handler is registered, an appropriate PMC is selected, and the PMC is configured to count the meaningful PMU events.

As discussed in Section 3.3.4, the PMU events of interest are `L3D_CACHE_ALLOCATE` for the A55 cores and `L2D_CACHE_WR` for the A76 cores. To reflect the worst-case scenario where a cache line read is always followed by a cache line write, the PMC is given a constant value for each regulation period:

$$set\_value = 0xffffffff - \frac{budget\_bw \cdot regulation\_period}{2 \cdot cache\_line\_size}, \tag{4.1}$$

where $0xffffffff$ is the maximum value for the 32-bit PMC, B_i is the assigned bandwidth for core i, P_i is the regulation period of core i, and the $cache\_line\_size$ is 64 Bytes for the A55 core and A76 core [14] [15].

If the read and write pattern of the application is known in advance, e.g. an application that only performs reads, the set value can be adjusted accordingly, e.g.

$$set\_value = 0xffffffff - \frac{budget\_bw \cdot regulation\_period}{cache\_line\_size}, \tag{4.2}$$

Once the PMC overflows, an interrupt is generated. The interrupt handling is illustrated in Algorithm 2. Jailhouse uses a generic interrupt handler for all the interrupts. In the generic interrupt handler, the interrupt number is queried from the GIC register `ICC_IAR1_EL1`. Each interrupt number represents a unique source of interrupt, so the specific handling is based on the interrupt number. The interrupt number of the PMU is 23 on the RK3588 [39]. If the generic interrupt handler has detected that the interrupt originates from the PMU, the PMU interrupt handler is called and it writes a `BLOCK` bit to the per-core public data structure in Jailhouse. Following that, the generic interrupt handler writes to the End of Interrupt (EOI) register `ICC_EOIR1_EL1` [52] to notify the GIC that interrupt has been handled and the core resumes to normal execution state. To enforce the CPU blocking, the `BLOCK` bit is checked after the EOI, and if it is set, the core is instructed to poll the timer value until the timer reaches the end of current regulation period. As the timer is inside the core, the polling does not consume DRAM bandwidth, which ensures that the core is blocked as expected.

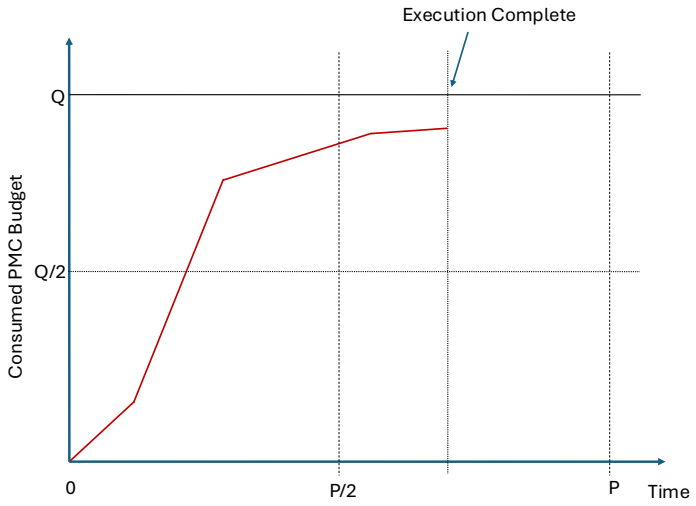
Algorithm 2: Interrupt Handler

```
1  generic_handler:
2      irq_number ← ICC_IAR1_EL1 /* Retrieve Interrupt Number */
3      if irq_number == 23 then
4          pmu_interrupt_handler();
5      end
6      if irq_number == 26 then
7          timer_interrupt_handler();
8      end
9      ICC_EOIR1_EL1 ← irq_number /* Signifies End of Interrupt */
10     memguard_cpu_block();

11 timer_interrupt_handler:
12     set_timer_next_compare_value();
13     cpu_public_data.memguard_block &= ~BLOCK;
14     set_pmc_budget(pmc_id, set_value);

15 pmu_interrupt_handler:
16     clear_pmc_overflow();
17     cpu_public_data.memguard_block |= BLOCK;

18 memguard_cpu_block:
19     if !(cpu_public_data.memguard_block & BLOCK) then
20         return
21     end
22     while !timer_reach_next_period() do
23         instruction_synchronization_barrier();
24     end
```

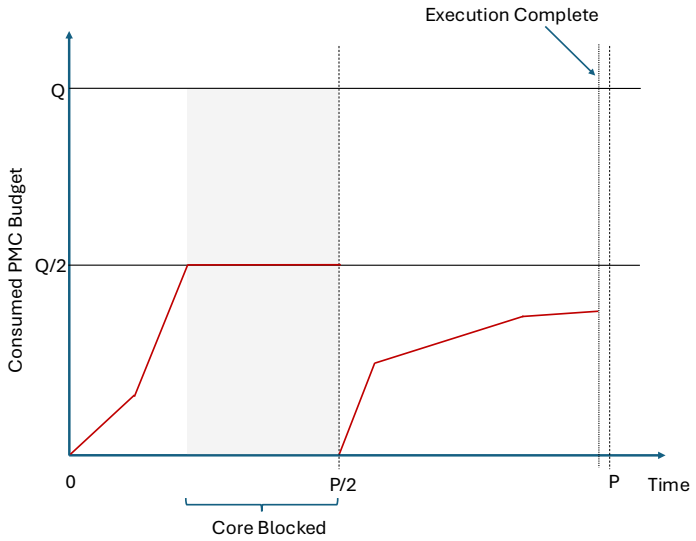
4.2.3 Regulation Period

The regulation period for each core is controlled by a per-core timer. On each A55 and A76 core, an EL2 Hypervisor physical timer [14] [15] is available. The timer will generates an interrupt as soon as its value reaches the given compare value while the timer is enabled and unmasked [65], which is rather straight forward. The timer count frequency is a system design choice, and therefore the memory bandwidth regulator should query the register `CNTFRQ_EL0` for timer frequency on the RK3588 [65].

For each regulation period, the timer generates an interrupt, which marks the beginning of the regulation period. As shown in Algorithm 2, the timer interrupt handler sets the interrupt for the next period, removes possible CPU blocking, and resets the PMC value.



(a) Long regulation period.



(b) Short regulation period.

Figure 4.1: Long and short regulation period in comparison. In the upper image, the regulation period is P, and the PMC budget is Q, while in the lower image, $\frac{P}{2}$ and $\frac{Q}{2}$ respectively. Both scenarios have the same memory bandwidth $B = \frac{Q}{P}$.

Regarding the length of a regulation period, a typical value is 1 ms [13] [21]. In fact, it is a configuration parameter and determined for each specific platform [13]. This will also be examined in Section 5.4. A shorter regulation period allows finer granularity control over the memory bandwidth, which is better for predictability, but also introduces more overhead as the context switch occurs more frequently. Moreover, the processing core may be throttled more frequently. In contrast, a longer regulation period effectively allows a larger amount of burst in memory access, as long as the budget within the period is not exceeded. By bursts in this thesis, it is meant sudden intensive accesses to the memory. A longer period

may therefore reduce the total latency where the CPU waits for the requested data to arrive, which is beneficial to performance. Figure 4.1 illustrates the difference qualitatively. A longer regulation period allows more bursts in memory access which may reduce the execution time, as shown in 4.1a. With a shorter regulation period, the regulator throttles the processing core more actively, as the gray area of Figure 4.1b shows. This may increase control granularity at the cost of performance. In the end, it is a trade-off to decide the optimal regulation period, as it depends on the tasks assigned to the processors and the acceptable interrupt overhead.

Chapter 5

Experiment and Evaluation

The memory bandwidth regulator is now evaluated on the RK3588. Since the regulator is intended for real-time purposes, the overall memory bandwidth that the DRAM system can provide reliably should be considered as the worst-case bandwidth, or maximum sustainable bandwidth. This chapter will first describe the experiment to determine the maximum sustainable bandwidth, and then explain the validation of the regulator. Further, the regulator is examined in the aspect of performance isolation, and overhead of the interrupts is measured.

5.1 Preparation

5.1.1 Benchmark

To test the regulator, then `bench` [66] and the San Diego Vision Benchmark Suite (SD-VBS) [67] are selected as the benchmarks for the experiment. `bench` is used to automatically determine the maximum sustainable bandwidth, while the SD-VBS has two most noteworthy benchmarks `disparity` and `tracking` as suggested by [21].

The `disparity` benchmark is more IO-intensive, which means that the performance measured by the benchmark is more likely to be bounded by the memory system. On the other hand, the `tracking` benchmark tends to burden the CPU more, and therefore it is compute-bound [21]. These two interesting benchmarks are later used to evaluate the performance isolation and the overhead of the regulator.

5.1.2 Measuring the Maximum Sustainable Memory Bandwidth

As mentioned in Chapter 1, the memory bandwidth varies greatly dependent on the access pattern and the state of the DRAM system. To ensure predictability, the DRAM system is always assumed to provide the maximum sustainable bandwidth. The `bench` triggers a number of write accesses and read accesses to stress the DRAM. It uses a variable step size to access the memory. For example, the first access is to the address `0xdeadbeef`, and the initial step size is the same as the cache line size which is 64 (`0x40`) (Byte). Then the next address to

access is `0xdeadbeef + 0x40 = 0xdeadbf2f`. With this step size, it keeps accessing the memory until the set time delay is elapsed. For the next, the step size is left shifted by one, which is `0x80`, and the benchmark stresses the memory with the step size. This continues until the step size is too large compared to the memory assigned to execute the benchmark.

There is always the specific step size where the memory bandwidth is the lowest, due to the characteristics of DRAM system. For the RK3588 in the thesis, the step size 262144 (1<<18, 256 KB) always causes the lowest memory bandwidth, which is also consistent with the result from [50].

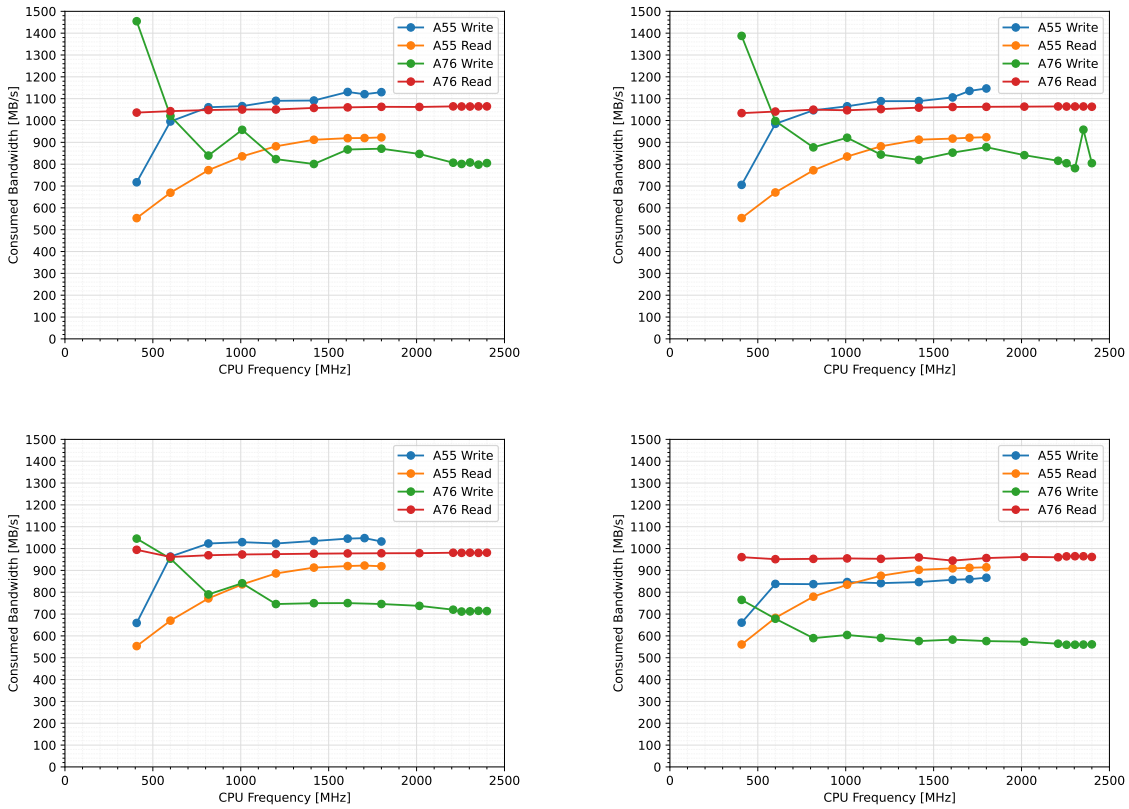


Figure 5.1: Measured memory bandwidth with different CPU scaling frequency and at different time. The A55 core read is the most stable case. The four figures are obtained at different sample time. The A76 core read rate does not change significantly over different CPU frequency, but it shows a difference of nearly 100 MB/s in bandwidth at different sample time. The write accesses of both A76 and A55 cores are the most unstable ones, and the cause is not yet clear.

However, the maximum sustainable memory bandwidth is somehow difficult to measure on the RK3588 with the current setup. As Figure 5.1 shows, the sustainable bandwidth fluctuates with regards to different CPU frequency[1]. Even worse, the measurement shows significantly different results at different moments. This could be caused by the Linux kernel (see Section 4.1.1) on the RK3588 or the firmware, whose DRAM scaling mechanism is not discovered yet and out of control. The maximum sustainable bandwidth is also measured in a Jailhouse inmate cell (Figure 5.2), and it is more stable and more consistent at different

[1]Similar behaviors were also observed in blogs: https://forum.radxa.com/t/rock-5b-debug-party-invitation/10483/472?u=tkaiser

measurement moments. For simplicity and testing, the maximum sustainable bandwidth was set to be roughly 900 MB/s, where the A55 core read stabilizes. The CPU frequency is set as a fixed value of 1800 MHz.

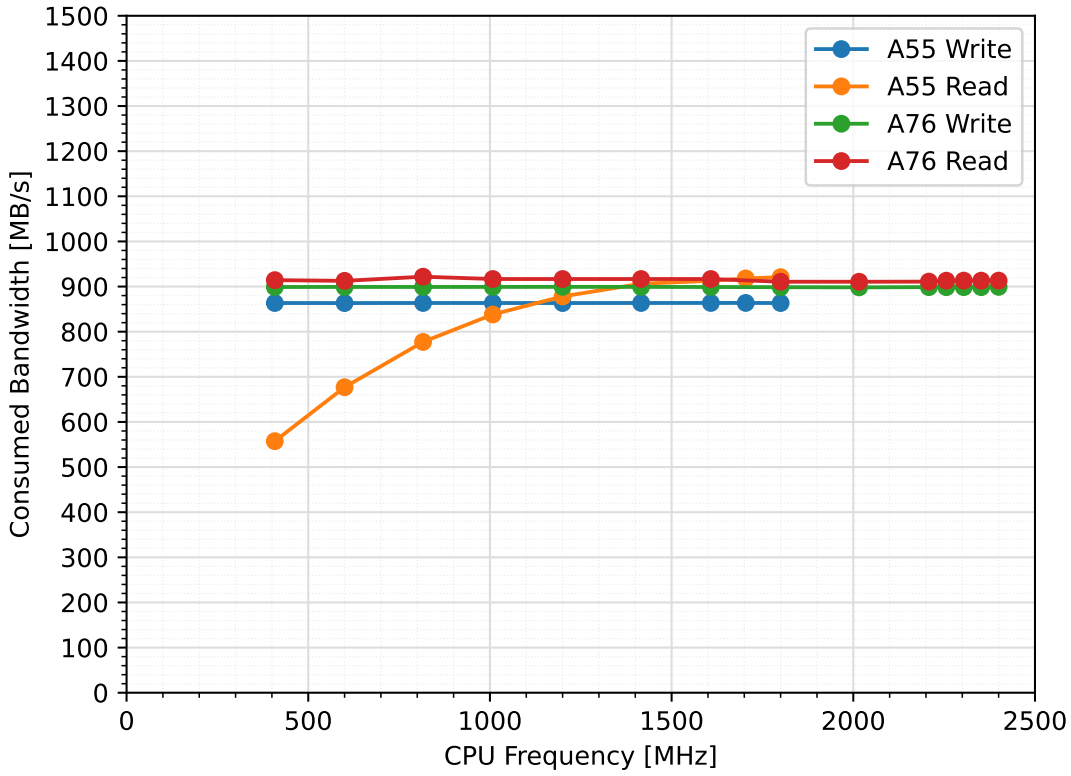


Figure 5.2: Measured memory bandwidth with different CPU scaling frequency in Jailhouse inmate cells.

5.2 Linear Model

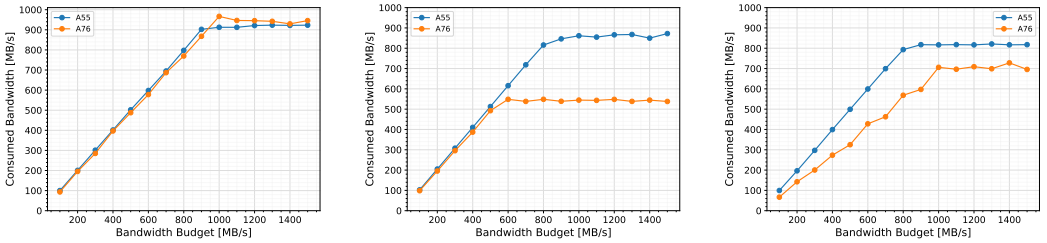
To validate the regulation executed by the regulator, the following experiments are conducted. Since both experiments intend to test whether the cores respect the given bandwidth by assigning bandwidth linearly, they are termed "linear model" in the thesis.

5.2.1 Regulation on one Single Core

If the memory bandwidth regulator operates correctly, the core should never consume more bandwidth than it is allowed. In the experiment, the examined core executes `bench` which issues worst-case reads, writes or modifies, while other cores are idle. The concepts of reads, writes, and modifies are explained in 3.3.4. It is noted that a write in the `bench` is always a full-line write, and thus does not cause the cache system to load (i.e., read) data from DRAM.

As shown in Figure 5.3, the core is assigned different amounts of memory bandwidth, and executes `bench` in isolation. The figure indicates that the consumed memory bandwidth on each core increases linearly with the assigned bandwidth budget, until the maximum sustainable bandwidth (see Section 5.1.2) is reached. The budget is never exceeded.

It should be noted that in Figure 5.3a and 5.3b, the optimistic Model 3.3 and Model 3.4 are used, while in Figure 5.3c the pessimistic Model 3.1 and Model 3.2 are used to estimate the per-core consumed memory bandwidth. As expected, given the more pessimistic estimation for the A76 cores, the behavior (especially for cache line modify) is below the given bandwidth budget, while the A55 cores are tracking more closely the given budget. Further investigation shows that Model 3.2 is too pessimistic especially with the step size between 16 KB to 1024 KB.



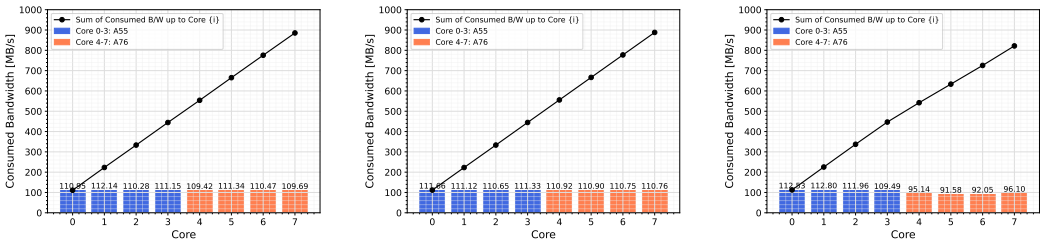
(a) Read access **(b)** Write access **(c)** Modify (read + write) access

Figure 5.3: The A55 and A76 cores attempt constant worst-case read or write access to the DRAM under bandwidth regulation respectively. Figure 5.3a shows the results where the core is doing reads constantly, and Figure 5.3b corresponds to the case where the core is doing writes. In Figure 5.3c, the core is doing modify. All the cores access memory with a 256 KB step size.

5.2.2 Regulation on multiple CPUs

In Section 5.2.1, a selected core executes the `bench` alone, which verifies the regulator design. In typical use cases, multi-core platforms execute tasks on multiple cores in parallel. To reflect those scenarios, the following experiment is conducted.

In this experiment (Figure 5.4), each core executes `bench` like Section 5.2.1, but this time, `bench` is executed in parallel on all cores. Each core is assigned 1/8 of the maximum sustainable bandwidth, which is 112.5 MB/s for each core. Same as in Section 5.2.1, the corresponding bandwidth estimation models are used. Figure 5.4a to Figure 5.4c indicate that each core can extract nearly the given bandwidth, and none of the cores exceeds the budget. In general, the sum of the consumed bandwidth increases linearly with the number of core accounted, which is also consistent with previous experiments in Section 5.1.2 and Section 5.2.1.

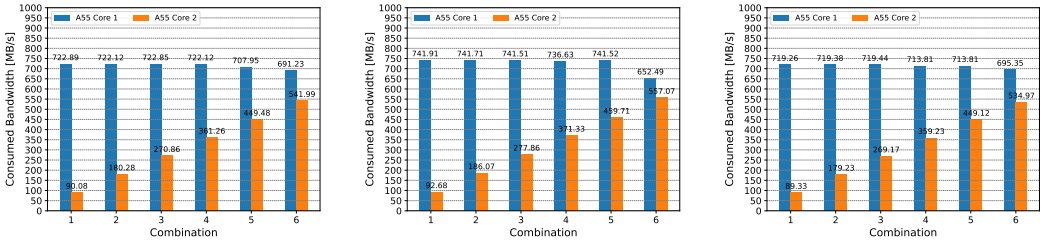


(a) Read access **(b)** Write access **(c)** Modify (read + write) access

Figure 5.4: The A55 and A76 cores attempt constant worst-case read or write access to the DRAM under bandwidth regulation in parallel. Figure 5.4a shows the results where the cores are doing reads constantly, and Figure 5.4b corresponds to the case where the cores are doing writes. In Figure 5.4c, the cores are doing modify.

5.3 Performance Isolation

Like MemGuard [13], the performance isolation effect of the regulator is also of interest. This feature is desired in real-time systems, because it means that the tasks do not, or only to the minimum extent, interfere with each other in terms of performance, i.e. execution time.



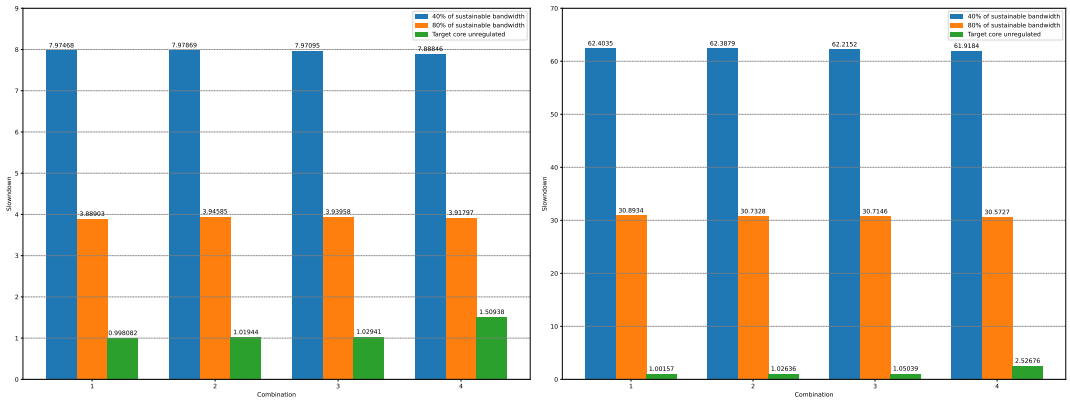
(a) Read access. **(b)** Write access. **(c)** Modify (read + write) access.

Figure 5.5: Two cores execute `bench` in parallel. One is assigned a fixed amount of bandwidth budget, roughly 80% of the maximum sustainable bandwidth, and the other one is assigned a variable amount of bandwidth budget, from 10% to 60%.

In Figure 5.5, two A55 cores are executing `bench` like previous experiments in Section 5.2.1 and 5.2.2. In each sub-figure, there are six combinations of bandwidth budgets assigned to the two cores. In every combination, core 1 is assigned a fix amount of bandwidth (80% of sustainable bandwidth). In comparison, core 2 has an increasing amount of bandwidth through different combinations, from 10% to 60% of sustainable bandwidth. It is shown that, as the assigned bandwidth budget of core 2 increases, the consumed bandwidth of core 2 grows accordingly. From the third combination in each sub-figure where core 1 has 80% and core 2 has 30% of sustainable bandwidth as budgets, the sum of assigned bandwidth budgets exceeds the maximum sustainable bandwidth. From then on, the assigned 80% to core 1 is no longer guaranteed, as core 2 is also stressing the DRAM system and hence may interfere the memory accesses from core 1. Though, the extracted bandwidth of core 1 may not decrease immediately, because there is parallelism in the DRAM system which may provide bandwidth higher than the sustainable bandwidth. For Figure 5.5a, the extracted memory bandwidth on core 1 begins to decrease when core 2 is assigned 50% of the sustainable bandwidth.

Figure 5.5b and Figure 5.5c show the same phenomena when core 2 is assigned 60% and 40% respectively.

Similar experiments are also conducted with the Disparity benchmark in the San Diego Vision Benchmark [67], as shown in Figure 5.6. Either a Cortex-A55 core (Figure 5.6a) or a Cortex-A76 core (Figure 5.6b) is selected as the target core to execute Disparity. There are four combinations of the assigned bandwidth budgets of the target core and seven co-runners. In each combination, the target core executes Disparity with assigned bandwidth budgets as marked by different colors, while the co-runners executes `bench` in parallel. In combination 1, all co-runners are idle, while from combination 2 to 4, each co-runner is assigned 40 MB/s, 60 MB/s, and unlimited amount respectively. The results indicate that our implementation of MemGuard is able to provide performance isolation. Meanwhile, it is clear that MemGuard can enforce significant slowdown if a small amount of bandwidth is assigned (blue bars in Figure 5.6). The assigned bandwidth is so small that the slowdown on the core is no longer sensitive to other cores. On the other hand, the performance of the target core is obviously interfered if no cores are regulated, as shown in the green bars of combination 4.



(a) Disparity on the A55 core.

(b) Disparity on the A76 core.

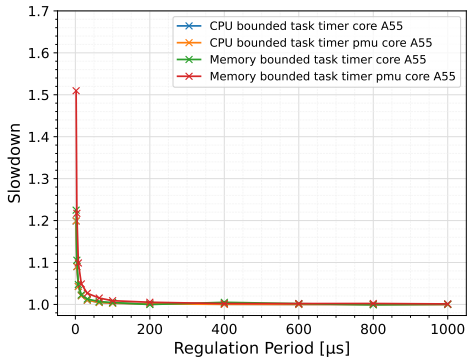
Figure 5.6: In 5.6a, a A55 core is executing Disparity while all other cores are running `bench` in parallel. In 5.6b, a A76 core is executing Disparity and all other cores are running `bench`. The figures show the slowdown of executing Disparity on the respective core. The reference baseline is the execution time when the core is not regulated and runs in isolation. In combination 1 of both figures, the co-runners are idle. From combination 2 to 4, each co-runner is assigned 40 MB/s, 60 MB/s, unlimited amount respectively. The colors correspond to different bandwidth budgets assigned to the core running Disparity, as marked in figures.

5.4 Overhead

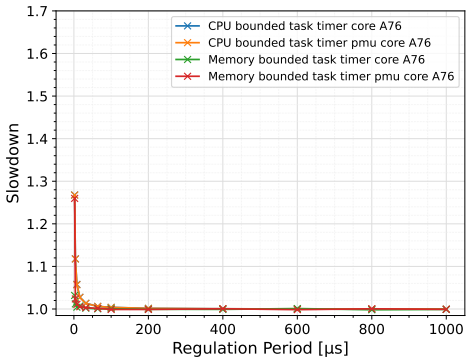
As mentioned in Chapter 4, the timer interrupt and PMU interrupt introduce overhead to the system. Particularly, the shorter the regulation period, the more frequently the interrupt occurs and consequently, the more overhead there will be. Therefore, the following experiment is conducted to quantify the overhead with regards to the regulation period on the cores.

In the test, the core under analysis executes either a CPU- or memory-intensive task, while the regulator runs in background. Same as the experiments before, the cores are

fixed at the frequency 1800 MHz. As the CPU-intensive task, the core executes square root calculation with certain number of iterations (in this thesis, `0x7fffffff` iterations), while as the memory-intensive task, the core executes `bench` with reads and step size 262144. Reference for CPU-bounded tasks is the execution time without memory regulation on the respective core. Reference for memory-bounded tasks is the highest bandwidth that the core can achieve without memory regulation and interference from other cores. It should be noted that the CPU throttling is disabled in the experiments, so as to reflect the overhead introduced by the interrupts themselves. In Figure 5.7, the PMU is configured to track the cache event as mentioned in 3.3.4, and it can be noticed that memory-intensive tasks have higher slowdown than CPU-intensive tasks especially when PMU interrupts are enabled (comparing the red curve and the green curve in Figure 5.7a and Figure 5.7b), because PMU interrupts are generated more frequently during memory-intensive tasks. In contrast, Figure 5.8 shows the results when the PMU counts CPU cycles.

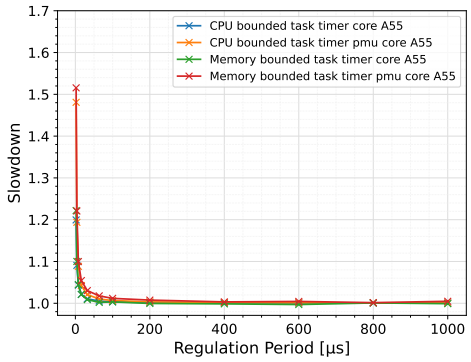


(a) Overhead on the Cortex-A55 core.

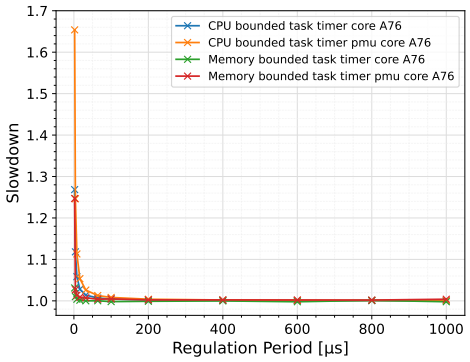


(b) Overhead on the Cortex-A76 core.

Figure 5.7: Overhead measurement on the Cortex-A55 and Cortex-A76 core. PMU is configured to count the cache events mentioned in Section 3.3.4. The PMC budgets are set to 1, so that it is very likely that the PMU interrupt is triggered.



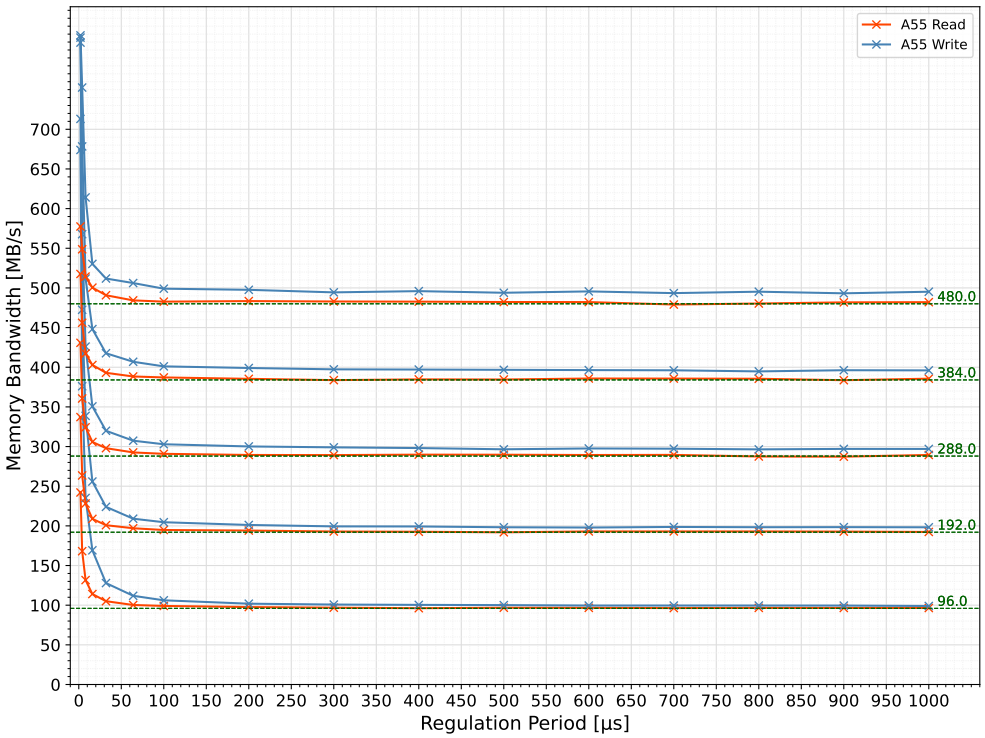
(a) Overhead on the Cortex-A55 core.



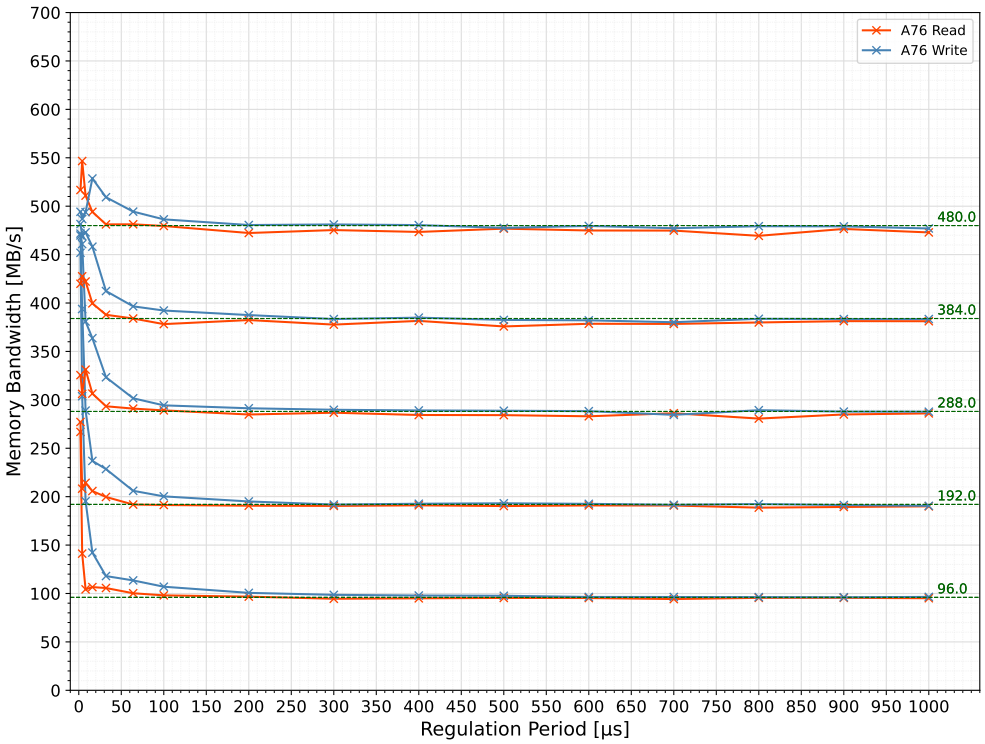
(b) Overhead on the Cortex-A76 core.

Figure 5.8: Overhead measurement on the Cortex-A55 and Cortex-A76 core. PMU is configured to count CPU cycles. The PMC budgets are set to 100 CPU cycles, so that the PMU interrupt is always triggered.

Compared to the MemGuard on ZCU102 where a slowdown is up to 2.4 at the regulation

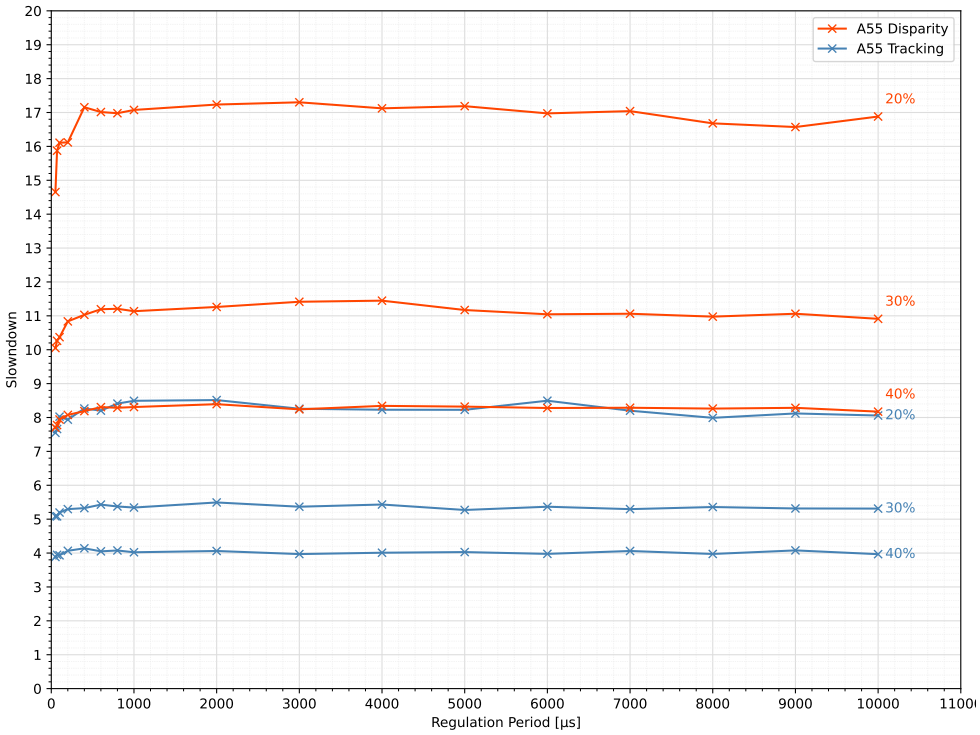


(a) Regulation error on the Cortex-A55 core.

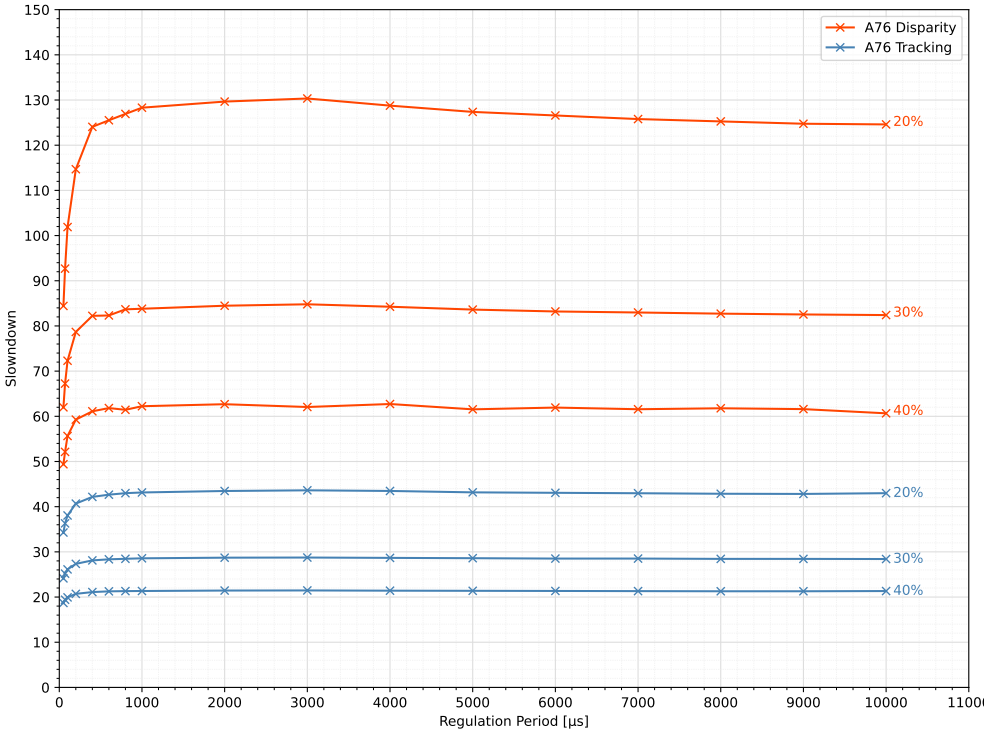


(b) Regulation error on the Cortex-A76 core.

Figure 5.9: Regulation error measurement on the Cortex-A55 and the Cortex-A76 core. The cores run `bench` with step size 262144. The horizontal green lines are the assigned bandwidth budgets respectively. The PMC budgets are adapted to the regulation period to keep the corresponding assigned bandwidth budgets. Shorter periods produce larger errors.



(a) Disparity and Tracking execution slowdown on the Cortex-A55 core.



(b) Disparity and Tracking execution slowdown on the Cortex-A76 core.

Figure 5.10: Disparity and Tracking execution slowdown of the cores. In each sample point, the benchmarks are executed ten times respectively and the average execution time is adopted. The percentage near the curves is the memory bandwidth budgets assigned to the processing core. The baseline is the execution time when the core is not regulated and runs in isolation. Note that the peak bandwidth of RK3588 is up to 22900 MB/s while the sustainable bandwidth is around 900 MB/s, and therefore the slowdown is much higher than the one on ZCU102 [21].

period of 32 μs [21], the regulator on the RK3588 exhibits very small slowdown ratio. According to the results in Figure 5.7 and Figure 5.8, the slowdown at regulation period 32 μs is maximally 1.03 for both the A55 and A76 core. Even at the 8 μs the slowdown is 1.10 for the A55 core and 1.07 for the A76 core respectively. The large difference could be related to the implementation of the regulator and the hardware. On the ZCU102, the overhead measurement is conducted with the version of MemGuard that is implemented on the Linux kernel [21], and the implementation at Linux kernel may increase the overhead for interrupt handling. In addition, the GIC on the RK3588 is v3 instead of v2 on the ZCU102, and this could also contribute to the difference.

5.5 Regulation Error

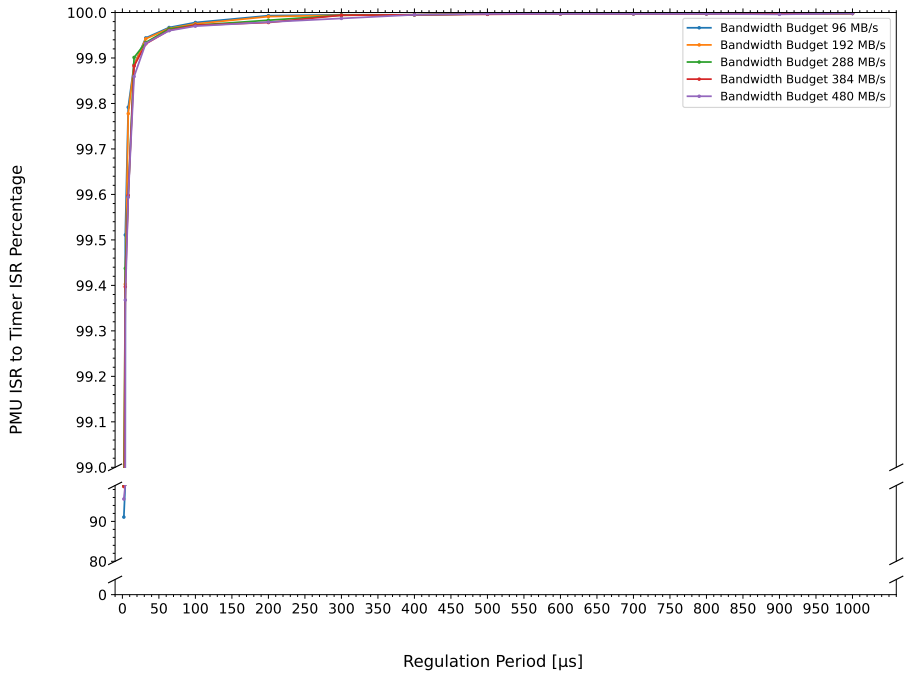
As discussed in Section 5.4, the overhead of shorter periods is very small compared to the ZCU102. Therefore, it is interesting to investigate the effects of shorter regulation periods. However, experiments show that with a short regulation period (shorter than 100 μs), this implementation of MemGuard seems to be unable to regulate the memory bandwidth. The regulation is still applicable with conventional periods, e.g. 1 ms, though.

Figure 5.9 shows the regulation error measured on the Cortex-A55 core and the Cortex-A76 core. `bench` is executed with reads and step size 262144 in the experiment. The horizontal green lines mark the assigned memory bandwidth respectively. It is shown in Figure 5.9 that on both A55 and A76 cores, given a memory bandwidth budget, the shorter the regulation period, the more the core uses bandwidth beyond the budget limit. Especially where the period is shorter than 100 μs, the bandwidth budget is exceeded drastically.

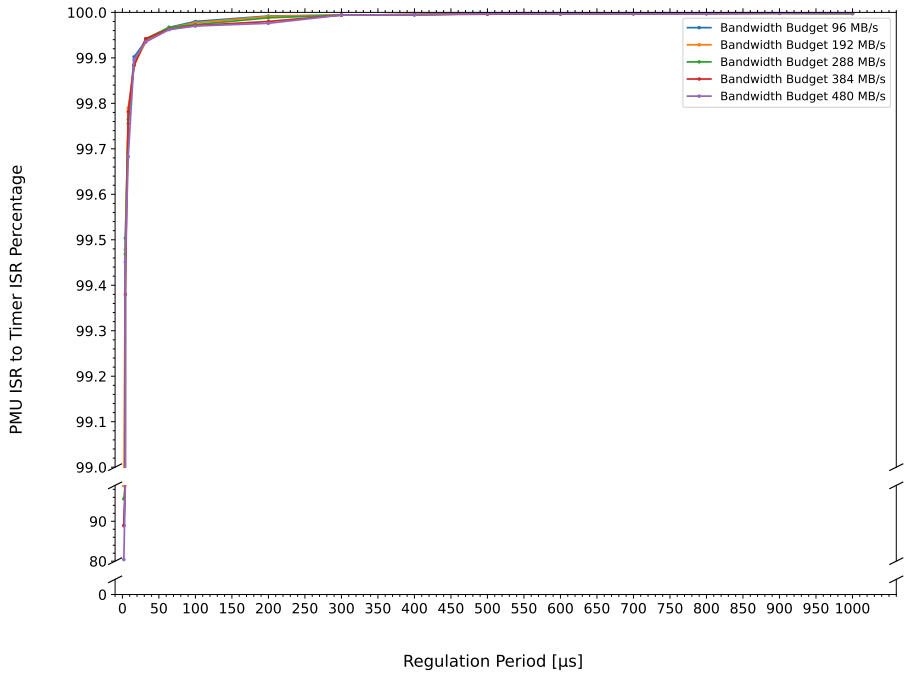
Similar results can be seen on the Disparity and the Tracking benchmark, as shown in Figure 5.10. For each combination of regulation period and bandwidth budget shown in the graph, the benchmarks are executed ten times, and the average execution time is calculated. The assigned memory budgets are marked by the percentages near the curves. As baseline, we use the execution time when the core operates without regulation and executes the benchmark in isolation. With a shorter regulation period, the slowdown is supposed to be higher, as the overhead becomes higher, as shown in [21]. In this experiment, however, the execution becomes faster at shorter periods. This means there are still some hidden mechanisms that destabilize the memory bandwidth regulator.

The cause of this regulation error is still not clear. A speculation of the direct cause is that the PMU interrupts are not triggered or handled properly. Figure 5.11 and 5.12 show the result of an experiment regarding the speculation. The core under analysis executes `bench` and is assigned different amounts of bandwidth budgets like previous experiments. The core is supposed to trigger a PMU interrupt almost in every regulation period, because it should consume all the bandwidth budget while executing this memory-intensive benchmark. In the experiment, the PMC values are set according to the regulation period to keep the corresponding assigned bandwidth budgets. At each sample point, `bench` and MemGuard execute in background for a specific time interval, which corresponds to roughly 100000

timer interrupts. The number of entries into the PMU interrupt handler (ISR) and the number of entries into the timer ISR are recorded.

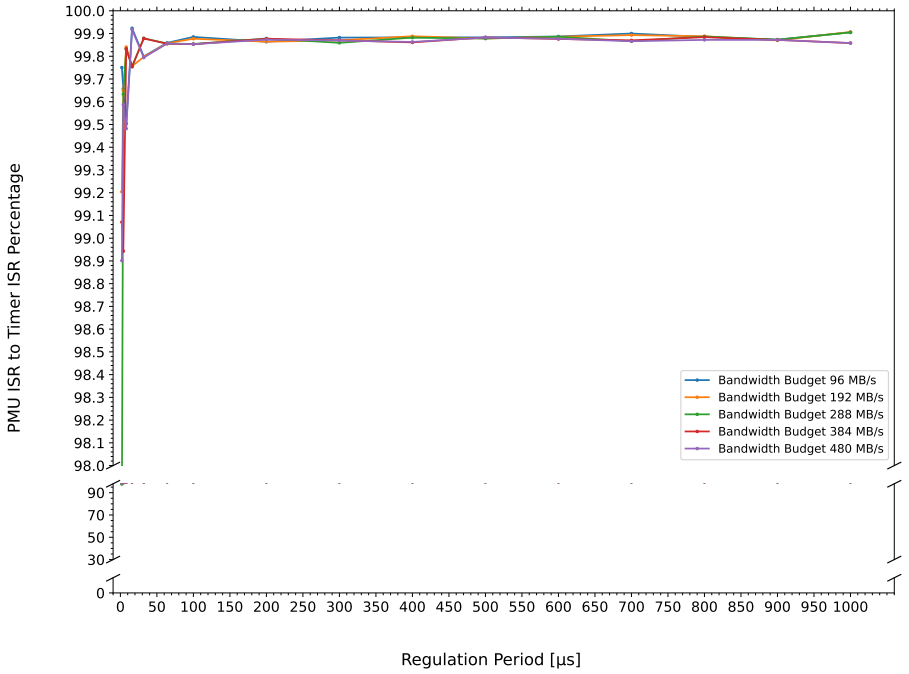


(a) Cortex-A55 Read.

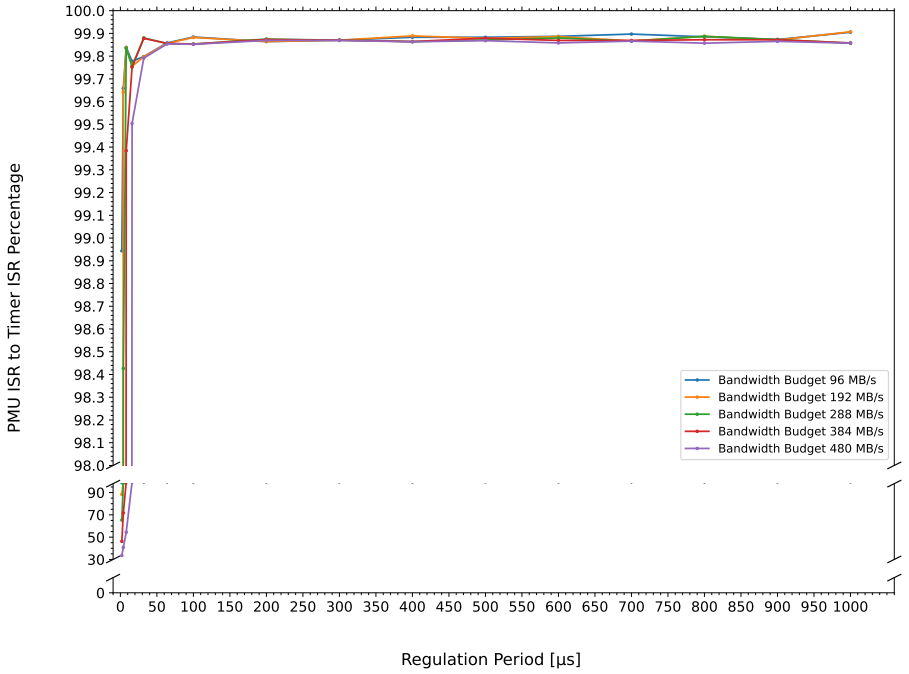


(b) Cortex-A55 Write.

Figure 5.11: The ratio of the number of PMU interrupt handler (ISR) entries to the number of timer ISR entries in percentage measured on the Cortex-A55 core. The number of timer ISR entries is fixed at roughly 100000 at each sample point. The PMU ISR is less entered at shorter regulation period.



(a) Cortex-A76 Read.



(b) Cortex-A76 Write.

Figure 5.12: The ratio of the number of PMU interrupt handler (ISR) entries to the number of timer ISR entries in percentage measured on the Cortex-A76 core. The number of timer ISR entries is fixed at roughly 100000 at each sample point. The PMU ISR is less entered at shorter regulation period.

The ratio of the number of entries into PMU ISR and number of entries into timer ISR is

shown in percentage in Figure 5.11 and Figure 5.12. It can be seen that when the regulation period is shorter, the PMU ISR is entered less frequently with regard to the timer ISR. Particularly at regulation period shorter than 100 μs, the ratio drops significantly. These results correspond to the experiment of Figure 5.9. Further investigation is needed.

Chapter 6

Conclusion and Limitations

In real-time systems, it is critical to meet the deadline of the task. Modern multi-core platforms introduce various sources of interference which impact the predictability of the system. This thesis introduces the background of interference, then focuses on the interference in the DRAM system, including DRAM, DRAM controllers, and DRAM bus. Different regulation techniques are analysed, and software memory bandwidth regulation (MemGuard) is adopted for the target platform RK3588, as this technique does not require modification to the hardware platform. In terms of memory bandwidth regulation, MemGuard and MemPol are state-of-the-art software memory bandwidth regulators. So far, these regulators have been implemented and evaluated only on x86 platform and on Arm platforms that do not feature the latest DSU capabilities. This work is the first to design, implement, and evaluate MemGuard for the RK3588 Arm-based platform. The RK3588 platform differs from previous platforms in various aspects, such as different CPU cores, the combination of performance cores and efficient cores, a different cache system with three levels of caches, and a different interconnect. This thesis analyses the platform and describes the porting of the Jailhouse hypervisor to it, as the preparation for designing and implementing the bandwidth regulator. Since RK3588 is one of the latest platforms, the Jailhouse hypervisor needs additional patches to be adapted for the platform. These include applying the patch [61] that allows SCMI-related SMC calls (Section 4.1.2) to reach the firmware, skipping registers that do not exist on the core, and disabling SDEI support of Jailhouse for easier MemGuard implementation logic. Furthermore, the Linux kernel on RK3588 requires patches [53] to enable Jailhouse and the kernel needs to be configured, such as disabling `CONFIG_KVM`, `CONFIG_ARM64_VHE` and `CONFIG_ARM_GIC_V3_ITS`. Considering the specification of the RK3588 platform, a memory bandwidth regulator with the same logic as the MemGuard on ZCU102 [31] [21] is designed and implemented. Compared to previous Cortex-A53-based implementations (e.g., on the ZCU102), the implemented MemGuard regulator adopts the model from [50] to regulate the memory bandwidth on different cores based on different PMU events available on the RK3588. Regarding the implementation, the Generic Interrupt Controller v3 is configured to enable and deliver PMU interrupts and timer interrupts to the corresponding cores. At last, extensive experiments prove that the MemGuard logic is still applicable on the state-of-the-art platform RK3588, with smaller execution overhead than previous platforms e.g. ZCU102.

Though, the memory bandwidth regulation system has some limitations. First, the regulator seems to malfunction at shorter regulation periods, as shown in Section 5.5, which limits the regulation granularity. Second, the maximum sustainable bandwidth is not accurate enough. The exact mechanism that destabilizes the sustainable bandwidth is yet unknown. Third, the memory bandwidth estimation models pessimistic, especially for the Cortex-A76 core. This may cause the memory bandwidth to be underutilized too much.

Bibliography

[1] Buttazzo, G. C. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Vol. 24. Springer Science & Business Media, 2011.

[2] Stankovic, J. A. and Ramamritham, K. *Hard real-time systems*. IEEE Computer Society Press, 1988. ISBN: 081868819X.

[3] Lugo, T., Lozano, S., Fernández, J., and Carretero, J. "A Survey of Techniques for Reducing Interference in Real-Time Applications on Multicore Platforms". In: *IEEE Access* 10 (2022), pp. 21853–21882. DOI: 10.1109/ACCESS.2022.3151891.

[4] Lee, E. A. and Seshia, S. A. *Introduction to embedded systems: A cyber-physical systems approach*. MIT press, 2016. ISBN: 978-0-262-53381-2.

[5] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., et al. "The worst-case execution-time problem—overview of methods and survey of tools". In: *ACM Transactions on Embedded Computing Systems (TECS)* 7.3 (2008), pp. 1–53. URL: https://doi.org/10.1145/1347375.1347389.

[6] Abella, J., Hardy, D., Puaut, I., Quinones, E., and Cazorla, F. J. "On the comparison of deterministic and probabilistic WCET estimation techniques". In: *2014 26th Euromicro Conference on Real-Time Systems*. IEEE. 2014, pp. 266–275. URL: https://doi.org/10.1109/ECRTS.2014.16.

[7] Sánchez-Puebla, M. A. and Carretero, J. "A new approach for distributed computing in avionics systems". In: *ISICT* 3 (2003), pp. 579–584.

[8] Abel, A., Benz, F., Doerfert, J., Dörr, B., Hahn, S., Haupenthal, F., Jacobs, M., Moin, A. H., Reineke, J., Schommer, B., and Wilhelm, R. "Impact of Resource Sharing on Performance and Performance Prediction: A Survey". In: *CONCUR 2013 – Concurrency Theory*. Ed. by D'Argenio, P. R. and Melgratti, H. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 25–43. ISBN: 978-3-642-40184-8.

[9] Ungerer, T., Cazorla, F., Sainrat, P., Bernat, G., Petrov, Z., Rochange, C., Quiñones, E., Gerdes, M., Paolieri, M., Wolf, J., Cassé, H., Uhrig, S., Guliashvili, I., Houston, M., Kluge, F., Metzlaff, S., and Mische, J. "Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability". In: *IEEE Micro* 30.5 (2010), pp. 66–75. DOI: 10.1109/MM.2010.78.

[10] Löfwenmark, A. and Nadjm-Tehrani, S. “Understanding shared memory bank access interference in multi-core avionics”. In: *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. 2016.

[11] Dasari, D., Akesson, B., Nélis, V., Awan, M. A., and Petters, S. M. “Identifying the sources of unpredictability in COTS-based multicore systems”. In: *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 2013, pp. 39–48. DOI: 10.1109/SIES.2013.6601469.

[12] Nagalakshmi, K. and Gomathi, N. “The impact of interference due to resource contention in multicore platform for safety-critical avionics systems”. In: *Int. J. Res. Eng. Appl. Manage.* 2.8 (2016), pp. 39–48. URL: https://www.ijream.org/papers/IJREAMV02I082020.pdf.

[13] Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., and Sha, L. “MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms”. In: *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2013, pp. 55–64. DOI: 10.1109/RTAS.2013.6531079.

[14] Arm Limited. *Arm Cortex-A55 Core Technical Reference Manual*. URL: https://developer.arm.com/documentation/100442/0200/?lang=en.

[15] Arm Limited. *Arm Cortex-A76 Core Technical Reference Manual*. URL: https://developer.arm.com/documentation/100798/0401/?lang=en.

[16] Rockchip Electronics Co., Ltd. *Rockchip RK3588*. URL: https://www.rock-chips.com/a/en/products/RK35_Series/2022/0926/1660.html.

[17] Rockchip Electronics Co., Ltd. *Rockchip RK3588 Brief Datasheet*. URL: https://www.rock-chips.com/uploads/pdf/2022.8.26/191/RK3588%20Brief%20Datasheet.pdf.

[18] Siemens AG. *Jailhouse Hypervisor*. 2023. URL: https://github.com/siemens/jailhouse/tree/next.

[19] Guo, D., Hassan, M., Pellizzoni, R., and Patel, H. “A Comparative Study of Predictable DRAM Controllers”. In: *ACM Trans. Embed. Comput. Syst.* 17.2 (Feb. 2018). ISSN: 1539-9087. DOI: 10.1145/3158208.

[20] Hennessy, J. L. and Patterson, D. A. *Computer architecture: a quantitative approach*. Elsevier, 2011. ISBN: 978-0-12-383872-8.

[21] Zuepke, A., Bastoni, A., Chen, W., Caccamo, M., and Mancuso, R. “MemPol: Policing Core Memory Bandwidth from Outside of the Cores”. In: *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2023, pp. 235–248. DOI: 10.1109/RTAS58335.2023.00026.

[22] Nesbit, K. J., Aggarwal, N., Laudon, J., and Smith, J. E. “Fair Queuing Memory Systems”. In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO’06)*. 2006, pp. 208–222. DOI: 10.1109/MICRO.2006.24.

[23] Rixner, S., Dally, W., Kapasi, U., Mattson, P., and Owens, J. "Memory access scheduling". In: *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*. 2000, pp. 128–138. DOI: 10.1145/339647.339668.

[24] Kim, H., Broman, D., Lee, E. A., Zimmer, M., Shrivastava, A., and Oh, J. "A predictable and command-level priority-based DRAM controller for mixed-criticality systems". In: *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. 2015, pp. 317–326. DOI: 10.1109/RTAS.2015.7108455.

[25] Valsan, P. K. and Yun, H. "MEDUSA: A Predictable and High-Performance DRAM Controller for Multicore Based Embedded Systems". In: *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*. 2015, pp. 86–93. DOI: 10.1109/CPSNA.2015.24.

[26] Mirosanlou, R., Hassan, M., and Pellizzoni, R. "DRAMbulism: Balancing Performance and Predictability through Dynamic Pipelining". In: *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2020, pp. 82–94. DOI: 10.1109/RTAS48715.2020.00-15.

[27] Guo, D. and Pellizzoni, R. "A Requests Bundling DRAM Controller for Mixed-Criticality Systems". In: *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2017, pp. 247–258. DOI: 10.1109/RTAS.2017.12.

[28] Ecco, L. and Ernst, R. "Improved DRAM Timing Bounds for Real-Time DRAM Controllers with Read/Write Bundling". In: *2015 IEEE Real-Time Systems Symposium*. 2015, pp. 53–64. DOI: 10.1109/RTSS.2015.13.

[29] Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., and Sha, L. "Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms". In: *IEEE Transactions on Computers* 65.2 (2016), pp. 562–576. DOI: 10.1109/TC.2015.2425889.

[30] Sohal, P., Tabish, R., Drepper, U., and Mancuso, R. "E-WarP: A System-wide Framework for Memory Bandwidth Profiling and Management". In: *2020 IEEE Real-Time Systems Symposium (RTSS)*. 2020, pp. 345–357. DOI: 10.1109/RTSS49844.2020.00039.

[31] Schwäricke, G., Tabish, R., Pellizzoni, R., Mancuso, R., Bastoni, A., Zuepke, A., and Caccamo, M. "A Real-Time Virtio-Based Framework for Predictable Inter-VM Communication". In: *2021 IEEE Real-Time Systems Symposium (RTSS)*. 2021, pp. 27–40. DOI: 10.1109/RTSS52674.2021.00015.

[32] Yun, H., Mancuso, R., Wu, Z.-P., and Pellizzoni, R. "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms". In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2014, pp. 155–166. DOI: 10.1109/RTAS.2014.6925999.

[33] Mancuso, R., Pellizzoni, R., Caccamo, M., Sha, L., and Yun, H. "WCET(m) Estimation in Multi-core Systems Using Single Core Equivalence". In: *2015 27th Euromicro Conference on Real-Time Systems*. 2015, pp. 174–183. DOI: 10.1109/ECRTS.2015.23.

[34] Mancuso, R., Dudko, R., Betti, E., Cesati, M., Caccamo, M., and Pellizzoni, R. "Real-time cache management framework for multi-core architectures". In: *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2013, pp. 45–54. DOI: 10.1109/RTAS.2013.6531078.

[35] Xilinx. *ZCU 102 MPSoC TRM*. URL: https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm.

[36] Arm Limited. *Arm Cortex-A53 MPCore Processor Technical Reference Manual r0p4*. URL: https://developer.arm.com/documentation/ddi0500/j/?lang=en.

[37] Arm Limited. *Arm DynamIQ Shared Unit Technical Reference Manual*. URL: https://developer.arm.com/documentation/100453/0400/.

[38] Arm Limited. *ARM CoreLink CCI-400 Cache Coherent Interconnect Technical Reference Manual*. URL: https://developer.arm.com/documentation/ddi0470/k/.

[39] Rockchip Electronics Co., Ltd. *Rockchip RK3588 Technical Reference Manual Revision 1.0*.

[40] Arm Limited. *ARM Cortex-A72 MPCore Processor Technical Reference Manual r0p3*. URL: https://developer.arm.com/documentation/100095/0003/.

[41] Perryman, N., Wilson, C., and George, A. "Evaluation of Xilinx Versal Architecture for Next-Gen Edge Computing in Space". In: *2023 IEEE Aerospace Conference*. 2023, pp. 1–11. DOI: 10.1109/AERO55745.2023.10115906.

[42] Arm Limited. *Learn the architecture - Introducing the Arm architecture*. URL: https://developer.arm.com/documentation/102404/0201/About-the-Arm-architecture.

[43] Arm Limited. *Learn the architecture - AArch64 Exception Model*. URL: https://developer.arm.com/documentation/102412/latest/.

[44] Subha, S. "A Few Processor Cache Architectures". In: *Management of Information Systems*. Ed. by Pomffyova, M. Rijeka: IntechOpen, 2018. Chap. 15. DOI: 10.5772/intechopen.77233.

[45] Danielsson, J., Seceleanu, T., Jägemar, M., Behnam, M., and Sjödin, M. "Testing Performance-Isolation in Multi-core Systems". In: *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. 2019, pp. 604–609. DOI: 10.1109/COMPSAC.2019.00092.

[46] Kloda, T., Solieri, M., Mancuso, R., Capodieci, N., Valente, P., and Bertogna, M. "Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems". In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2019, pp. 1–14. DOI: 10.1109/RTAS.2019.00009.

[47] Martins, J. and Pinto, S. "Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems". In: *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2023, pp. 40–53. DOI: 10.1109/RTAS58335.2023.00011.

[48] Zhang, Z., Liu, Y., Chen, J., Qi, Z., Zhang, Y., and Liu, H. "Performance Analysis of Open-Source Hypervisors for Automotive Systems". In: *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*. 2021, pp. 530–537. DOI: 10.1109/ICPADS53394.2021.00072.

[49] Patterson, D. A. and Hennessy, J. L. *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann, 2016. ISBN: 978-0-12-801733-3.

[50] Jiang, Y. "Analysis of Cache and Memory Interference on Arm DSU Systems". MA thesis. Munich: Technical University of Munich, 2024.

[51] Arm Limited. *Learn the architecture - Introducing CoreSight debug and trace*. URL: https://developer.arm.com/documentation/102520/latest/.

[52] Arm Limited. *Learn the architecture - Generic Interrupt Controller v3 and v4, Overview*. URL: https://developer.arm.com/documentation/198123/latest/.

[53] Siemens AG. *Siemens Linux*. 2021. URL: https://github.com/siemens/linux/tree/jailhouse-enabling/5.10.

[54] Shenzhen Xunlong Software Co., Ltd. *Orange Pi 5 Plus*. URL: http://www.orangepi.org/html/hardWare/computerAndMicrocontrollers/service-and-support/Orange-Pi-5-plus.html.

[55] Shenzhen Xunlong Software Co., Ltd. *Orange Pi Linux*. URL: https://github.com/orangepi-xunlong/linux-orangepi/tree/orange-pi-5.10-rk3588.

[56] Shenzhen Xunlong Software Co., Ltd. *Orange Pi 5 Plus User Manual*. URL: http://www.orangepi.org/html/hardWare/computerAndMicrocontrollers/service-and-support/Orange-Pi-5-plus.html.

[57] Siemens AG. *Bootstrapping the Partitioning Hypervisor Jailhouse*. 2016. URL: http://events17.linuxfoundation.org/sites/events/files/slides/ELCE2016-Jailhouse-Tutorial.pdf.

[58] Arm Limited. *Learn the architecture - Generic Interrupt Controller v3 and v4, LPIs*. URL: https://developer.arm.com/documentation/102923/0100/LPIs.

[59] Arm Limited. *Arm System Control and Management Interface Platform Design Document*. URL: https://developer.arm.com/documentation/den0056/latest/.

[60] Arm Limited. *SMC Calling Convention (SMCCC)*. URL: https://developer.arm.com/documentation/den0028/f/?lang=en.

[61] Ramsauer, R. *s32g3 HACK: Allow SMIC passthru in root cell*. URL: https://github.com/lfd/jailhouse/commit/3a88b0b371aeb649bc496d8c272b5d3ab5de3982.

[62] Sinitsyn, V. *Understanding the Jailhouse hypervisor, part 1*. URL: https://lwn.net/Articles/578295/.

[63] Arm Limited. *Software Delegated Exception Interface (SDEI) Platform Design Document*. URL: https://developer.arm.com/documentation/den0054/c/?lang=en.

[64] Arm Limited. *ARM Generic Interrupt Controller Architecture version 2.0 - Architecture Specification*. URL: https://developer.arm.com/documentation/ihi0048/latest/.

[65] Arm Limited. *Learn the architecture - Generic Timer*. URL: https://developer.arm.com/documentation/102379/0103/What-is-the-Generic-Timer-.

[66] Zuepke, A. *bench*. URL: https://gitlab.com/azuepke/bench.

[67] Venkata, S. K., Ahn, I., Jeon, D., Gupta, A., Louie, C., Garcia, S., Belongie, S., and Taylor, M. B. "SD-VBS: The San Diego Vision Benchmark Suite". In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 2009, pp. 55–64. DOI: 10.1109/IISWC.2009.5306794.