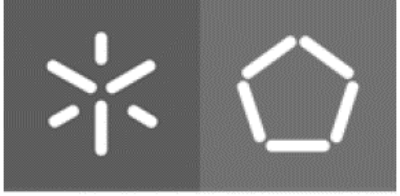
Universidade do Minho
Escola de Engenharia

Francisco Manuel Barreto Rocha

Mitigating Platform-Level Memory Interference on a Static Partitioning Hypervisor

dezembro de 2022



Universidade do Minho
Escola de Engenharia

Francisco Manuel Barreto Rocha

Mitigating Platform-Level Memory Interference on a Static Partitioning Hypervisor

Dissertação de Mestrado
Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Sandro Pinto

dezembro de 2022

Agradecimentos

Gostaria de agradecer a algumas pessoas que me apoiaram e acompanharam durante todo o meu percurso académico, que sem elas este percurso teria sido muito mais difícil, se não impossível:

Gostaria de agradecer à minha família, à minha mãe, ao meu pai, à minha irmã e ao meu irmão, por me terem dado a possibilidade de ter este percurso académico mais facilitado.

À minha namorada, Inês Saraiva, por me ter acompanhado em todos os momentos deste percurso e pelo apoio incondicional. Agradeço a enorme compreensão e a contribuição que teve na minha chegada ao fim deste percurso.

Aos meus amigos, David Correia, Marcelo Amaral, João Peixoto, Rafael Cachetas, André Pereira e Rui Esteves por estes cinco longos anos de batalha.

Ao meu amigo, João Rodrigo, em especial, por realmente me ter aturado e por tudo o que passamos para chegar aqui. Sempre disponível a ajudar, sempre que tinha algum problema que me desse mais dores de cabeça.

Gostaria também de agradecer ao meu Orientador, professor Sandro Pinto, pela possibilidade da concretização de um tema tão desafiante e fora do meu conhecimento, assim pelo voto de confiança. Também gostaria de agradecer ao José Martins, doutorando na qual trabalhei na sua tese, pelo conhecimento que me passou e por tudo no seu geral.

Por fim, o meu profundo e sentido agradecimento a todas as pessoas que contribuíram para a concretização desta dissertação, estimulando-me intelectual e emocionalmente.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration. I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

Os sistemas embebidos apresentam uma enorme heterogeneidade, incluindo desde pequenos controladores, para sistemas inteligentes operados a bateria até sistemas de computação de alto desempenho em automóveis. Definitivamente, os sistemas embebidos cresceram exponencialmente nos últimos anos, aumentando o seu nível de complexidade numa multiplicidade de indústrias. Tal facto, aportou naturalmente uma consolidação de várias camadas de critícidade sobre a mesma plataforma.

Para proporcionar um aumento de isolamento e segurança, a indústria começou a recorrer às capacidades da tecnologia de virtualização. A tecnologia de virtualização permitiu a consolidação e o isolamento de múltiplos subsistemas numa mesma plataforma. Hipervisores, também designados de monitores de máquinas virtuais, são responsáveis pela gestão dos recursos de hardware para as mesmas.

No entanto, os hipervisores tradicionais não foram desenhados para garantir requisitos de tempo-real e para as restrições embebidas. Nesse sentido, tanto a indústria com a academia direcionaram esforços para uma nova arquitectura de virtualização, os chamados hipervisores de partição estática. Um exemplo destes hipervisores é o Jailhouse que aloca estáticamente o hardware no momento de inicialização. No entanto, este hipervisor exige que o sistema operativo Linux o inicialize, o que se traduz em desvantagens em termos de tempo de arranque e segurança.

Para abordar a maioria dos problemas do Jailhouse, o nosso grupo de investigação desenvolveu o Bao. Bao é um hipervisor estático que não apresenta qualquer dependência externa. Este mesmo hipervisor já implementa a partição da cache de modo a mitigar as interferências causadas pela contenção da partilha da mesma entre máquinas virtuais. Contudo, existem outros pontos de contenção na hierarquia de memória. Assim, a primeira parte desta dissertação concentra-se sobre a implementação de um mecanismo para regulação da largura da banda de memória, cujo o objectivo é o isolamento temporal do controlador DRAM. O mecanismo implementado irá ser avaliado para identificar o nível de capacidade de redução de interferência na DRAM. A segunda fase desta dissertação, explora os possíveis casos de "side-channel" através do uso de DMAs para avaliar os casos de interferência a nível da plataforma e não apenas ao nível do CPU.

Abstract

Embedded systems range from tiny controllers in batery-operated smart devices to high-performance computing systems in cars. Therefore, embedded systems have grown exponentially in recent years in various industries, and this has lead to increased complexity, that naturally pushed towards consolidation of several layers of criticality on the same platform.

To provide increasing isolation and security, industry started to leverage the capabilities of virtualization technology. Virtualization technology enable the consolidation and isolation of multiple subsystems onto the same platform. Hypervisors, also called as Virtual Machine Monitors are responsible for managing the hardware resources for different virtual machines (VMs).

However, traditional hypervisors are not designed for real-time requirements and embedded constraints. This lead academia and industry to work towards a new virtualization architecture, the so called static partitioning hypervisors. These hypervisors statically allocate hardware resources. An example of a static partitioning hypervisor is Jailhouse, which statically allocates hardware at initialization time to the different VMs. However, this hypervisor requires the Linux operating system to bootstrap itself, leading to several bottlenecks in terms of boot time and security.

To address most of the issues of Jailhouse, our research group has developed Bao. Bao is a static hypervisor that does not have any external dependencies. The Bao hypervisor already implements cache partitioning to mitigate interference caused by the contention in the shared cache between guests. However, there are other points of contention downstream of the memory hierarchy. So the first part of this dissertation focuses on implementing a mechanism to regulate VM memory bandwidth, where the goal is the temporal isolation of the Dynamic Random Access Memory memory controller. The implemented mechanism will be evaluated to identify the extent to which the mechanism helps solve the interference problems in the DRAM.

The second phase of the dissertation explores possible cases of side-channels through the use of Direct Memory Accesss to evaluate cases of interference at the platform level and not just at the Central Processing Unit level.

Contents

List of Figures

List of Listings

List of Tables

List of Algorithms

Glossary

APB Advanced Peripheral Bus

ARM Advanced RISC Machine

ATF Arm Trusted Firmware

AXI Advanced eXtensible Interface

BD Buffer Descriptor

CCI Cache Coherent Interconnect

CPU Central Processing Unit

DMA Direct Memory Access

DRAM Dynamic Random Access Memory

FPD Full Power Domain

GEM Gigabit Ethernet Mac

GIC Generic Interrupt Controller

HVM Hardware Virtual Machine

I/O Input/Output

IoT Internet of Things

IPA Intermediate Physical Address

IPC Inter-Process Communication

LLC Last Level Cache

LPD Low Power Domain

MMU Memory Management Unit

MPAM Memory System Resource Partitioning and Monitoring

OS Operating System

PA Physical Address

PMC Performance Monitoring Counter

PMU Performance Monitoring Unit

PPI Private Peripheral Interrupt

PTE Page Table Entry

SMMU System Memory Management Unit

TBU Translation Buffer Unit

TCB Trust Code Base

TCU Translation Control Unit

TLB Translation Lookaside Buffer

VA Virtual Address

vCPU Virtual Central Processing Unit

VM Virtual Machine

VMM Virtual Machine Monitor

1. Introduction

Embedded systems are microprocessor-based computer systems, usually built into a system or product, with a dedicated operational role. They range from small controllers in smart home devices to avionics systems in airplanes to large networking switches that make up our telecommunication networks. Thus, embedded systems have grown exponentially in various industries, such as automotive, robotics, and industrial automation, in recent years [1]. This growth has led to a steady increase in the number of requirements for the past few years, as being pressured to minimize size, weight, power, and cost has made it increasingly necessary to design real-time embedded systems to integrate different levels of criticality on a common hardware platform [2, 3]. With the growth of these systems, the concern for isolation between sub-systems has also increased. Thus, virtualization technology has emerged as a natural response to provide such security.

The main focus of virtualization is to allow the co-existence of multiple guests on the same hardware platform while guaranteeing spatial and temporal isolation between them [4]. This technology has become a key enabling technology in the server domain, thereby providing benefits such as reducing the cost associated with the purchase, set up, cooling, and maintenance. With technologies like hypervisors, it is possible to separate the physical resources from the virtual environments. Hypervisors have been traditionally split into two types 2.8 [5]:

- type one - bare-metal hypervisors that run guest virtual machines directly on a system's hardware, essentially behaving as an operating system, making this type the most commonly used;
- Type two - hosted hypervisors behave more like traditional applications that can be started and stopped like an ordinary program. The hypervisor itself runs on top of an operating system.

Using a hypervisor makes it possible to manage the hardware resources, virtualizing the resources assigned to virtual machines or guests. In this way, the guest will feel that it is using the hardware, but it will be using a virtualized copy, so the hypervisor's job is to use the actual hardware [6, 7]. However, traditional hypervisors such as KVM or Xen were not originally designed for the real-time requirements and embedded constraints [8, 9].

Static partitioning hypervisors is an architecture that leverages hardware-assisted virtualization to employ a thin layer of software [10]. This architecture statically partitions all the platform resources and assigns them to the existing VMs. By statically partitioning the hardware, it allows for more robust isolation

and real-time constraints. In this type of architecture, there is a one-to-one mapping between virtual CPUs and physical CPUs, thus is not required a scheduler.

A hypervisor of this type is Jailhouse, this hypervisor has the disadvantage that the whole system relies on a Linux to start the entire system, which is an issue for critical systems with stringent boot time requirements.

Thus, a lightweight, in-house, open-source hypervisor called Bao emerged, designed primarily to tackle mixed-criticality systems, with multiple software stacks running in parallel with each other, as illustrated in Figure 1.1. Bao allows to run an user interface on a Linux alongside a safety-critical application that it controls. It aims to provide strong isolation and real-time guarantees. Unlike many hypervisors, it does not have any external dependencies [10].

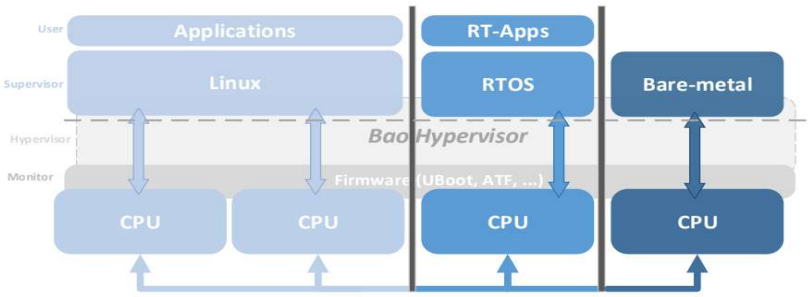


Figure 1.1: Bao's static partitioning architecture.

One of the problems in multicore platforms that are managed by a hypervisor is the fact that many guests may share hardware, even if a dedicated core is assigned to each domain, memory accesses by these guests interfere with one another. This type of problem has a significant impact on system performance, and in the case of real-time systems, the operation of the system is also affected because it depends on time constraints.

Although Bao already provides cache partitioning via cache coloring and static partitioning of CPUs, there is still interference in the DRAM controller because this resource is still shared between guests. The DRAM interference problem exists because if a guest makes large and continuous accesses to the memory, it will interfere with the executions of the other guests. The main problem to be solved is bandwidth regulation. To this end, a memory bandwidth reservation mechanism (memory throttling) will be implemented for the temporal isolation of the DRAM controller. The application of this technique must be implemented for each domain and not for each CPU.

The second phase of the dissertation is to explore possible interference cases, either through the use of DMAs, device sharing, anything that can cause interference at the platform level and not only at the CPU level. This is because there is little information about platform-level interference.

1.1 Objectives

This dissertation aims to provide the hypervisor Bao with better temporal isolation in what concerns the DRAM. The implementation will be made in an Advanced RISC Machine (ARM)v8 architecture in a ZCU104 platform. The main objectives for the dissertation are:

1. Analyse solutions for memory interference for the Bao hypervisor regarding the DRAM;
2. Implement a bandwidth regulation mechanism in Bao to reduce memory interference;
3. Analyse memory interference at platform level in Bao hypervisor, more in specifically when using DMA devices.

1.2 Document Structure

The remaining of this dissertation is structured as follows. Chapter two overviews the fundamental concepts and ideas of virtualization and how it is currently exploited in computing systems. It is then followed by an explanation of basic concepts in ARM processors and virtualization. In the end, it is given a thorough explanation of the problem and possible solutions, as well as related work showing possible implementations for the existing solutions. Chapter three goes trough the system specification, as well as an overview on the benchmarks utilized. Chapter four explains the design, implementation, evaluation, and results of the memory throttling mechanism. Finally, but not least, chapter five, where it is explained in detail every scenario evaluated and the conclusions taken from the tests to study the platform-level interference.

2. Background and State of the Art

This chapter will present the fundamental concepts needed for the development of this project and some existing related work relevant to the dissertation's theme. The concepts described in this background focus on Arm architecture but still applied to other architectures. In the beginning, the basics will be explained to understand the problem. Afterward, it will be explained and contextualized virtualization and how this is currently exploited in computing systems, followed by a breakdown of the problem and possible solutions. The end presents related work that shows possible mechanisms to resolve specific problems.

2.1 Computer Architecture

Computer architecture describes computer systems' functioning, structure, and implementation by a set of guidelines and procedures. That defines how a computer's hardware components are interconnected and how data is transferred and processed. Also, the definitions of architecture can state a description of a computer's capabilities and programming paradigm rather than a specific implementation. Thus, computer architecture involves instruction set architecture design, microarchitecture design, logic design, and implementation.

2.1.1 Cache

A cache is a small amount of fast, on-chip memory between the core and main memory. Its job is to hold copies of items in the main memory. It is a reserved storage location that collects temporary data, which makes it easy to quickly retrieve frequently used information helping devices to run faster. The cache size is small, considering the overall memory used in the system, because larger caches make for more expensive chips compared to DRAM blocks, which have more capacity [11]. Whenever the core wants to read or write to a specific address, it first looks for this address in the cache. If it finds what it is looking for in the cache, it uses the data instead of making more extended access to the main memory to retrieve this information.

Modern ARM processors are usually implemented with two or more levels of cache [11]. That is a small L1 Instruction and Data cache and a larger, unified L2 cache. In addition, there can also exist an external L3 cache. The L2 and L3 caches do not distinguish between program data and program instructions. In Figure 2.1 it is illustrated an example of a typical cache with two levels of cache.

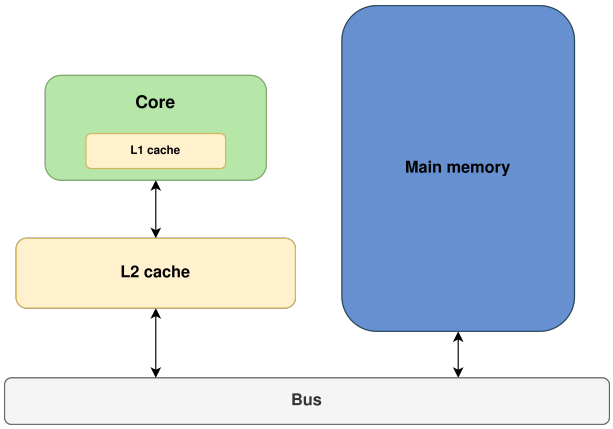


Figure 2.1: Illustration of a generic multi-level cache organization.

2.1.2 Memory Translation

ARM uses a virtual memory system where the virtual address used by code is translated to a physical address. This process is run by the Memory Management Unit (MMU). The MMU uses page tables stored in memory to translate the addresses, which the MMU reads when necessary. This process is known as the table walk.

In a non-hypervisor system, in which the software is running directly in the hardware, it is only needed a stage-1 (single-level) page table translation to translate the virtual address (VA) to a physical address. However, in a hypervisor system, guests in each VM use their stage-1 page table, but this first stage does not translate to the actual physical address but translates to the intermediate physical address (IPAs). This translation corresponds to the addresses in the IPAs space. Requiring another stage for the translation to the actual physical address (PA), as illustrated in Figure 2.2. So the hypervisors have a second layer of page tables, stage-2, that maps the IPAs to the physical address. These two stages of translations ensure that each VMs can only access the hardware assigned to them.

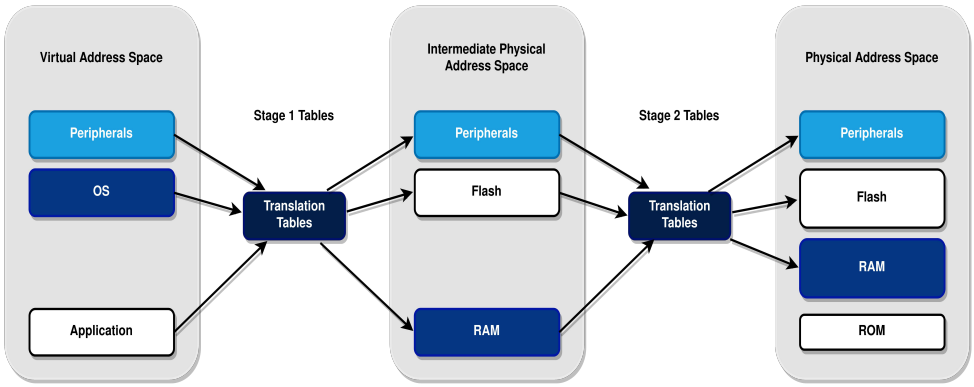


Figure 2.2: Memory translation with stage-2.

Translation Lookaside Buffer (TLB)

In an Operating System (OS) that uses virtual memory, every process believes it works with a large, contiguous memory section. However, the memory may be dispersed throughout the physical memory or moved to secondary storage. Thus, for each process will be created a page table. A page table is where the OS will store its mappings of Virtual Address (VA) to Physical Address (PA), which will contain Page Table Entry (PTE) [12]. The PTE will contain information like frame numbers and other useful bits. The frame number addresses the main memory where the page table is residing. Using such a technique to support virtual memory can lead to high-performance overheads [12]. This overhead is because each table needs to be consulted before each memory access. The big problem is that the mapping information is generally stored in the physical memory under normal conditions, which causes the processor to need more accesses to the memory, in fact, because there are multiple levels of page tables, and if it is stage-2, each level will be needed to be translated leading up to sixteen accesses for page-tables of three levels. Thus, to help speed up the process, it is used a hardware-accelerated solution. A Translation Lookaside Buffer (TLB) is used to speed up the address translation, making the TLB fundamental in accessing memory.

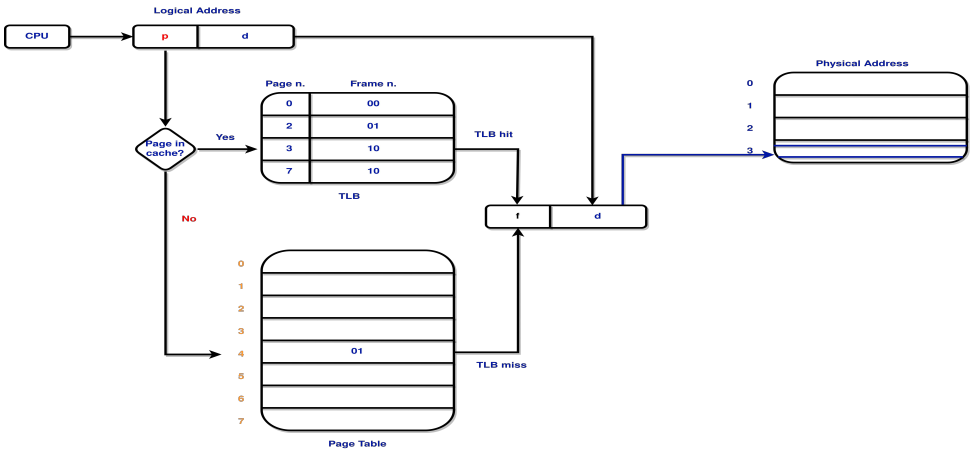


Figure 2.3: Translation Lookaside Buffer example.

A TLB is a high speed cache and is part of the chip's memory-management unit. The TLB's purpose is to keep track of recently used transactions. It contains the PTE that has been used most recently. Upon each virtual memory reference, the hardware first examines if the translation is present in the TLB (TLB hit), and if so, the translation is made quicker and without the need to access the page table that contains all the translations. So the TLB improves the memory access speed considerably when the necessary data are not found in the L1 and L2 caches. If a PTE is not found in the TLB (TLB miss), the page number is used as an index to search for the page, as illustrated in Figure 2.3. The MMU checks if the page is already in the main memory. If not, a page fault is set, and then the OS hypervisor is responsible for updating the page table so that next time the MMU searches for the same page, it founds the translation in the page table.

2.1.3 Performance Monitoring Unit

Using profiling tools, developers can assess how their code interacts with the CPU. A program's analysis may be distilled using the information on cache access, memory reads/writes, and processor clock cycles.

So, the ARM processor provides a Performance Monitoring Unit (PMU) as part of its architecture to enable the gathering of processor execution info. Using the PMU, it is possible to track events occurring on the core via counters. The PMU allows the recording of architectural and micro-architectural events for profiling purposes. To enable this feature, it will be necessary to configure the PMU to select events that will increment the counter. A list of supported events is possible to obtain by accessing the ARM manual [13] and their respective event numbers, which the PMU can track. It is also essential to stand out that the PMU allows to set a threshold for a specific counter, and when this counter reaches the threshold defined, an interrupt is generated. Section 4.2.2 gives a more detailed description.

2.1.4 System Memory Management Unit

The ARM System Memory Management Unit (SMMU) is a hardware component that performs address translation, and access control for bus initiators outside of the CPU [14]. The SMMU has the same functions as the Memory Management Unit (MMU). The only difference is that the SMMU is not connected to the CPU but instead to various hardware accelerators, for instance, DMA that has direct access to the memory. The DMA might be allocated for use by a VM. The access made by the DMA goes through the SMMU [15].

In a non-hypervisor system, the DMA is programmed via a driver, usually, in kernel space [16]. The driver then has specifications so that one application cannot use the DMA to access memory regions not permitted.

In a hypervisor system, stage-2 is extended to cover other hardware accelerators, for example, the DMA controller. This is because every access to the memory made by an application is monitored through the MMU. If a violation is made, the application is terminated. However, accelerator devices do not have the same restrictions when accessing the memory. This allows them to access any part of the memory space, causing security issues, corrupting data, and modifying another application's memory. To prevent this from happening, it needs additional hardware to sit between the accelerator and the memory system, requiring an MMU, and this MMU is referred to as SMMU [16]. When a VM allocates a DMA it has direct access to the memory, and without the SMMU when the VM interacts with the DMA, creates the following issues:

- The first problem is isolation because the DMA is not subjected to the 2-stage of memory translation, and so it has direct access to the PA;

- The second problem is that the VM believes the PAs are IPAs. And so, without using the SMMU, the memory view from the DMA and the VM are entirely different. In order to overcome this problem, the hypervisor would need to trap the interactions between the VM and the DMA to translate the addresses.

The hypervisor is responsible for the SMMU programming so that the DMA upstream can see the same as the VM in which the DMA is assigned, as illustrated in Figure 2.4. With the use of SMMU is possible to enforce isolation between the VMs, ensuring that these hardware accelerators can not be used to breach memory areas compromising others VMs. The streamID is used to uniquely identify the master that initiated a transaction. Each master has a streamID associated that is propagated in the bus. The streamID maps the incoming transaction to a context by using the stream mapping table.

A typical SMMUv3-based system includes multiple TBUs. Each Translation Buffer Unit (TBU) is located close to the component that it provides address translation for [17]. The TLB, if available, is used by the TBU to deliver the necessary translation after intercepting transactions. The TBU asks the Translation Control Unit (TCU) for translations when the TLB does not already have them, then caches the translation in one of the TLBs.

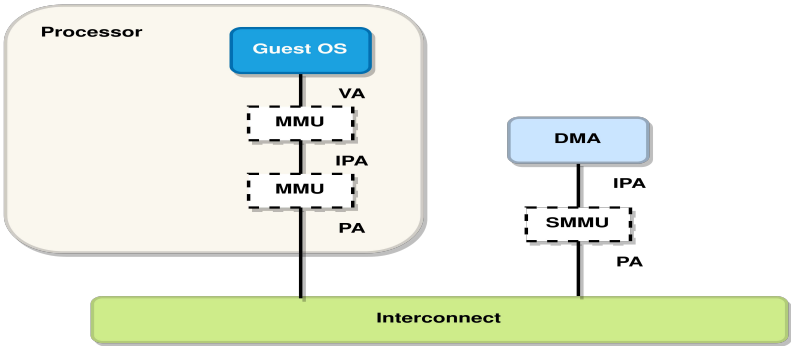


Figure 2.4: SMMU location on a systems topology.

SMMU Performance Monitoring Extension

The SMMU Performance Monitors Extension is an memory-mapped extension. Like the CPU PMU but now regarding the SMMU. The Performance Monitors Extension's inclusion in an SMMU is optional. The register map for the Performance Monitors registers is unknown if it is not supported.

The SMMU Performance Monitors Extension offers event counter resources and event filtering based on a translation context or a StreamID if implemented. The address map of a translation context bank reveals event counter resources.

- **StreamID groups**

  The idea of a StreamID group is included in the SMMU Performance Monitors Extension. A group of StreamIDs is known as a set of StreamIDs. Event counters are associated with a StreamID group. Only events brought on by the processing of transactions related to that group can be counted by an event counter. Event counters have a defined affiliation with a StreamID group. There is no way to alter the connection. A StreamID is affiliated with a counter group specifically.

- **Event Filtering**

  A Counter group counts events based on the global, translation context, or StreamID:

  - **Global Basis** - It takes into account all transactions related to the StreamID group that the Counter group belongs to;

  - **Translation Context Basis** - It solely takes into account transactions handled by the SMMU translation context identified by a particular register and constrained by the scope of the StreamID group to which the Counter group belongs.

  - **StreamID Basis** - Only transactions that match the ID and mask assigned by a specific register are taken into account, and their scope is constrained by the StreamID group to which the Counter group belongs.

2.2 Virtualization

Virtualization is a broad term in computer science, where it uses software to create an abstraction layer over computer hardware. The software layer providing this environment is referred to as Virtual Machine Monitor (VMM) or hypervisor. Virtualization can also be considered a high consuming technique since its usage requires a lot of disk space and RAM, apart from adding a management layer to the stack [5].

This technique allows the hardware elements of a single computer, for instance, processors, memory, storage, and more, to be divided into multiple virtual computers, commonly called virtual machines (VMs) [18, 19]. Each VM can run either full-fledged OSes or bare-metal applications. It behaves like an independent computer, even though it runs on just a portion of the underlying computer hardware, thus allowing each VM acting as guests, to run in an isolated environment, where the hypervisor is responsible for the scheduling of each VM and conciliate the hardware access. The hypervisor acts as the host as it runs in a higher privilege level than the VMs, as they run on top of it, as illustrated in Figure 2.5. Furthermore, any resources used by each VM are controlled by the virtualization environment, which means that even if a guest wants to exceed the resource limitations, it is impossible since the environment restricts them [20, 21].

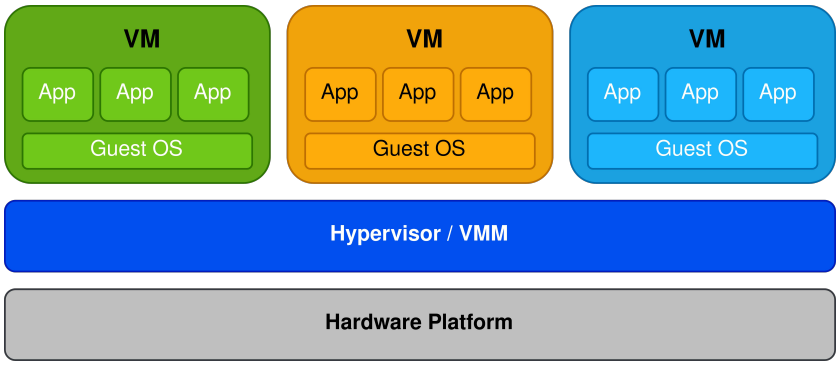


Figure 2.5: System virtualization stack.

In the past, embedded systems were just computer systems that carried out a small number of tasks focused on dedicated functions that the system needed to perform with low complexity of the software [5]. However, this scenario is changing, and modern embedded systems are moving towards general-purpose systems each day, which means that nowadays, it is needed for embedded systems to perform a large number of tasks, for instance, users or devices can interact with them in a variety of ways to meet a broad range of needs. Because of this, their functionality is proliferating, driven by the complexity and size of the software needed which is also growing, for instance, the development of artificial intelligence (AI), machine learning, and the Internet of Things (IoT) [22]. Most of these scenarios are general purpose applications applied to embedded systems and the use of applications written by developers that have little or no acknowledge at all about the embedded constraints.

Although embedded systems bring characteristics typical of general-purpose systems, traditional differences between them still exist. They remain real-time systems and still have the same resource constraints, for instance, the power availability in the form of a battery, resulting in a tight energy budget [4], since these devices are supposed to operate through several hours or even days without any battery recharge. A strong motivation to adopt virtualization is the safety that it provides [4]. Nowadays, it becomes more likely for an application from an OS to get compromised, resulting in a system-wide security breach. This type of breach can be reduced by running such OSes in a virtual machine, limiting the OSes access to the machine.

2.2.1 Full-Virtualization

Until now the general term virtualization was discussed to make mention of the classical technology, also known as full-virtualization [18]. In full-virtualization, guests are hosted unmodified and are unaware of their virtualization, and sensitive OS calls are captured and translated using binary translation. This happens whenever the virtual board attempts to execute a privileged instruction, for instance, I/O request, memory write and others, which causes a trap into the hypervisor, as illustrated in Figure 2.6. The hypervisor provides each VM with all the services of the physical system, including virtual devices, and virtualized memory management. The virtualization layer fully disengages the guest OS from the underlying

hardware. Nevertheless, full-virtualization techniques have strict requirements, such as those of classical virtualizability or the need for hardware support. Microsoft Hyper-V and Parallels systems are examples of full virtualization [23].

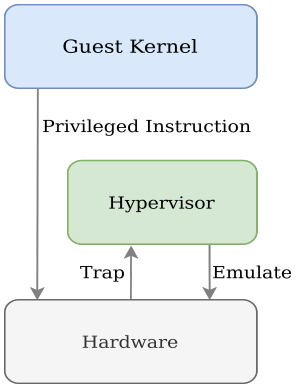


Figure 2.6: Full-Virtualization stack.

2.2.2 Para-Virtualization

Contrarily to full-virtualization exists another concept called para-virtualization. Para-virtualization enhances virtualization technology in which a guest is modified prior to installation inside a virtual machine (VM). This modification will allow the guest OS to adapt to the virtual machine environment, replacing any critical or sensitive instructions to invoke the hypervisor (hypercalls), meaning that instead of sending system calls intercepted by the hypervisor, they send deliberate system calls to specific subsystems, as illustrated in Figure 2.7. It is worth mentioning that the OS's source must be available to make such customization [20]. The main limitation of para-virtualization is that the guest OS must be customized specifically to run on top of the virtual machine monitor (VMM), that is, the host program that allows a single computer to support multiple, this allows each guest to know that it is running on top of the VMM and not on top of the hardware. Para-virtualization eliminates the need for the virtual machine to trap privileged instructions. Trapping is handling unexpected or unallowable conditions that can be time-consuming and can adversely impact performance in the systems that employ full virtualization. VMware Workstation Pro and Xen are some examples of para-virtualization [20, 23].

Full-virtualization and para-virtualization can imply large overheads because manipulating instructions must be intercepted and rewritten. Thus, it has become more common to exist virtualization support to hardware so that the hypervisor can delegate more efficiently the access to the protected resources. This hardware-assisted virtualization comes with promises to fix the performance issues, and flexibility [8].

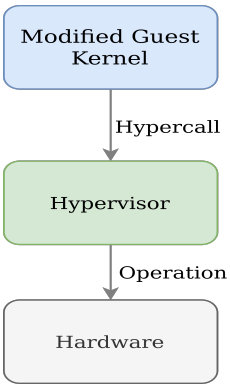


Figure 2.7: Para-Virtualization stack.

2.2.3 Hardware Virtualization

ARM Virtualization

The ARM virtualization is set to enable hypervisors to run multiple unmodified OSes on the same platform. Using a provided standard hardware implementation it is possible to create high-performance hypervisors [24]. The ARM virtualization extension provides each OS an illusion of sole ownership of the entire system by introducing a new set of architectural features:

1. **Hypervisor mode**: This mode is set in the EL2, which is expected to be the hypervisor software managing multiple guest OSes in the EL1 and EL0. Furthermore, the hypervisor mode only exists in the non-secure world.

2. **Additional memory translation**: ARM virtualization includes a second translation level for memory addresses, called stage-2. The first stage translates VA to Intermediate Physical Address (IPA) and then from IPA to PA. The OS can configure the first stage without being aware of virtualization. The hypervisor can manage the second stage.

3. **Interrupts**: The interrupts can be configured to be rerouted to the hypervisor, and then the hypervisor is responsible for taking that interrupt to the corresponding guest.

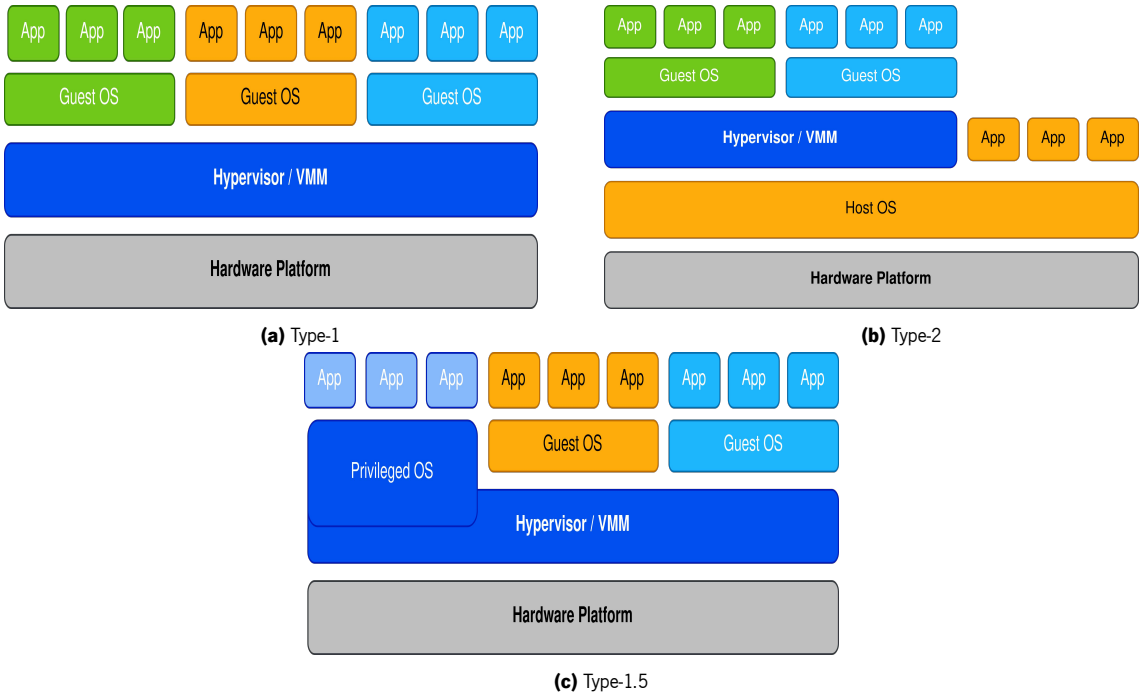
4. **Hypervisor call**: ARM virtualization includes instructions to implement hypercalls so that the guests can request hypervisor services.

2.2.4 Hypervisor

Hypervisors serve as an interface between the VM and the underlying physical hardware, ensuring that each VM has access to the physical hardware it needs, and so it must not result in noticeable performance degradation. One of the challenges of hypervisors is securely multiplexing several virtual machines over a

single set of physical resources [6]. Hypervisors also ensure that the VMs do not interfere by accessing each other's memory space or computing cycles. Hypervisors can be classified as one of three types, as illustrated in Figure 2.8[5]:

- Type-1 or "bare-metal" hypervisors, also known as native hypervisors, have direct access to the hardware and run in a higher privileged mode than their guests. As a result, the performance of the guest is influenced by the performance of the hypervisor. In general, this makes this type of hypervisor perform better and more efficiently.
- Type-2 or hosted hypervisors run as a software layer on top of the OS of the host machine. Usually, this type does not have permission to perform any operations on the hardware directly since the responsibility rest on the OS below the hypervisor. As a result, the performance of this type of hypervisor is much lower than the bare-metal hypervisors.
- Type-1.5 is a combination of type-1 and type-2, the performance of type-1, and the power of an entire OS provided by type-2.



(a) Type-1 **(b)** Type-2

(c) Type-1.5

Figure 2.8: Virtualization Topologies

Bear in mind that we have been referring to type-1 hypervisors until this point. It is preferable because of its independence from the host OS. In addition, this type of hypervisor is a more secure option. Contrary to the hosted hypervisor, they do not depend on the underlying OS. Thus, for the same reason, it generates less overhead, and any malfunction in an individual VM does not harm the rest of the system. This type of topology will be implicit throughout the dissertation since this topology meets the embedded requirements.

Architectures

Now will explained the more common hypervisor architectures, which are the monolithic and microkernel architectures. Although all hypervisors perform the same tasks, for instance, starting and maintaining virtual machines, abstracting system resources, and sharing hardware, their design differs to meet specific performance requirements, annul some weaknesses, and adjust to specific characteristics of the problem.

Monolithic

Monolithic hypervisors are contained in a thin layer of software that supports virtualization, which condenses all the subsystems within a single entity. This entity includes the hardware drivers, application program interfaces, and the virtualization stack [7]. Besides this, the hypervisor also needs the essential modules to provide the virtual machine abstraction, such as memory isolation, managing CPU multiplexing, and others [18]. The VMs are allocated directly above the hypervisor, making the job of the virtual machine monitor interact with each guest. The hypervisor exists entirely in single layer, which means that all of its trust code base (TCB) resides in the most secure area, thus bringing a disadvantage since the TCB of the monolithic architecture is very large, and in some of the most used OSes, it can reach millions of lines of code. All this code runs with complete access to all systems resources, which means that all this code must be trustworthy. The problem is that such an extensive system comes with strings attached, like bugs and vulnerabilities. Since this type of architecture condenses all subsystems within a single layer and with this type of risk complementing, one fault in one of the subsystems can bring the entire system down. The monolithic architecture supports hypercalls, and a risk that comes with the use of hypercalls is their misuse when transitioning to para-virtualization. In conclusion, monolithic systems are not the most secure and trustworthy [18][7].

Microkernel

The term "microkernel" was coined in response to the predominant monolithic kernels at the time [6]. Microkernel architectures have a smaller footprint than the monolithic since they are designed to minimize code in the hypervisor space [7]. These are developed to have a minimal amount of software to provide the mechanisms needed to implement an OS. Traditional operational system functions, such as device drivers, protocol stacks, and file systems, are typically removed from the microkernel and run in userspace. The microkernel is solely responsible for the essential services only, and so it keeps the most critical services inside the kernel, and the rest are present in the userspace, as illustrated in Figure 2.9. Regular communication between program/application is instituted through message passing, thus reducing the speed of execution of microkernel. This type of design also allows isolation between user and kernel services. If any of the user services fail, it will not affect the functioning of kernel services. Another advantage is that if it becomes necessary to add a new user service to the system, it is easily added, hence no need to modify the kernel space. As stated in [6] from a research perspective, it is easier to

work on a single system component if that component is not entangled with other code. As the monolithic architecture, the microkernel also supports hypercalls.

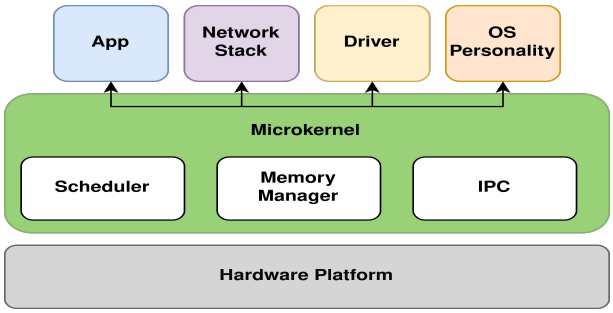


Figure 2.9: Microkernel Architecture.

Traditional Hypervisors

There are a variety of traditional hypervisors, but the most common ones are KVM, and Xen. These hypervisors run on bare-metal and support multiple VMs. They also need to provide device drivers and other components required to support complete full virtualization.

KVM (for Kernel-based Virtual Machine) is a hardware-assisted virtualization that is an extension of the Linux kernel's execution mode to allow its operation as a hypervisor. KVM is a monolithic hypervisor that supports a full-virtualization environment for its guests and provides para-virtualization to support I/O when using VirtIO subsystems. When using VirtIO, the I/O is handled in the user space, primarily through QEMU. VirtIO is a virtualization standard for device drivers where the guest's device driver is aware of running in a virtual environment and communicates directly with the hypervisor. Unfortunately KVM was not design for embedded requirements, like a small footprint and real-time guarantees [9].

Xen is an open-source hypervisor platform sponsored by Citrix. Xen hypervisor is slightly modified to be integrated into cloud solutions from Citrix [7]. The Xen hypervisor is a type-1.5, microkernel hypervisor [7]. Xen supports two different types of guests that can be used simultaneously on a single Xen system. The guests either run with a particular modified OS referred to as para-virtualization (PV) or unmodified OSes leveraging special virtualization hardware called hardware virtual machine (HVM).

It uses para-virtualization extensively to achieve very high performance, although it requires modifications to the guest OS [25]. Xen already implements hardware-assisted virtualization. Using this configuration, the guest OSes do not need to be modified, making it possible to use, for example, Windows guests in the Xen hypervisor [8].

The HVM guests do not have the front-end drivers as the PV, as these are not modified and require emulations. To boost performance, fully virtualized HVM guests can use special paravirtual device drivers to bypass the emulation for disk and network IO, called qemu-dm.

Embedded Hypervisors

The embedded hypervisors target more security and real-time performance differently from the traditional hypervisors. Although traditional hypervisors are widely used in enterprise servers, the embedded world continues to ramp up. Traditionally hypervisors are used to provide virtualization and some level of isolation, and significant problems of virtualization include execution speed, security, memory management, multiplexing, and isolation of devices. However, another option to traditional hypervisors is microkernel hypervisors that originated the embedded hypervisors, and a part of the microkernel is specifically designed to provide isolation and security [21]. Some examples of embedded hypervisors are XVISOR, and ACRN. Some conditions need to be met in the embedded hypervisor to have efficient virtualization:

1. **Isolation**: The VMs need to be isolated from the hypervisor and each other. This safety needs to be enforced without the hypervisor knowing anything about the software from the guest.
2. **Performance**: The performance of the software running in the VM can only show a minor decrease compared to running directly on the hardware;
3. **Real-time**: It must provide high-bandwidth and low-latency communication between the system's hardware and/or system components, such as the CPU and the emulator.

Static Partitioning Hypervisors

Static partitioning hypervisors follow an architecture that leverages the hardware-assisted virtualization to employ a thin layer of software, very different from other hypervisors that need a much larger TCB [26]. This architecture statically partitions all the platform resources and assigns them to existing VMs. From a hardware point of view, it is still shared, but from the software point of view, it is separated into different slots, from where the guest can only reach its assigned hardware [10]. By statically partitioning the hardware allows for more robust isolation and real-time guarantees. Each virtual CPU is statically assigned to a single physical CPU, not needing a scheduler, decreasing the size and complexity of the system. Some examples of static partitioning hypervisors are Jailhouse and Bao.

Jailhouse is a hypervisor enabled by a kernel module from within a fully booted Linux system [27]. After the startup code is executed, the hypervisor takes over the hardware resources and lifts the Linux into a virtual machine, as illustrated in Figure 2.10, reassigning the hardware back to the Linux guest accordingly to a configuration of the system. Thus, the jailhouse hypervisor acts as VMM. The Linux initially is used as a boot loader for the hypervisor but not as an OS. A disadvantage that this brings is that the system relies on a Linux to start the system, having problems with boot-time and still possibly having a relative big code footprint [28].

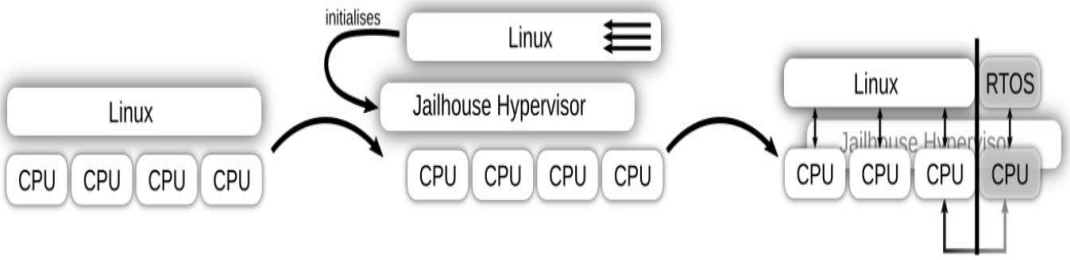


Figure 2.10: Demonstration of how the start-up of Jailhouse works.

Bao is a lightweight bare-metal hypervisor. It comprises a thin layer of privileged software to implement its static partitioning architecture [26]. Bao targets ARMv8 architecture and experimental support for the RISC-V architecture. Bao is designed for mixed-criticality systems (MCS) by providing strong isolation and real-time guarantees. The only external dependence on this hypervisor is the standard platform management firmware [3].

Bao's memory is allocated statically at initialization time, the virtual machines' IO is limited to pass-through, the virtual interrupts are directly mapped to the physical interrupts, and the virtual CPUs are assigned to the physical ones using a 1:1 mapping. The hypervisor allows for intercommunication between VMs through a static shared memory and inter VM interrupt triggered through a hypercall [10].

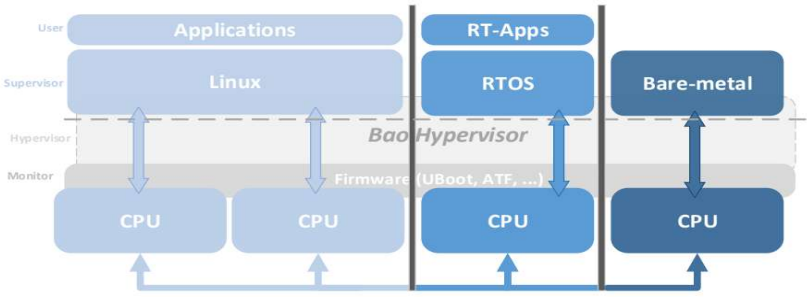


Figure 2.11: Bao's static partitioning architecture.

Bao has three principles:

- **Minimalism and Simplicity**: The aim is to maintain a minimal and straightforward code base. The hypervisor is therefore only implemented in architectures that provide hardware-assisted virtualization;
- **Least Privilege**: Every system component should only have access to what is absolutely essential. The only information that each core may access is specific to the VM assigned to it. The hypervisor cannot directly access the physical memory of the VMs;
- **Thorough Isolation**: Though virtualization offers isolation, VMs still engage with one another. Such is interference caused by the L2 cache, which the several VMs share. For this reason, Bao

employs a cache coloring mechanism (sub-section 2.3.1). With this approach, the cache is segmented by colors, and Bao can be set up such that several VMs use various portions of the same cache, improving isolation. Nevertheless, it causes various issues, such as memory fragmentation.

2.3 Spatial and Temporal Isolation on Multicore Platform

To increase the performance of CPUs, the computer industry shifted from single-core to multicore platforms in modern computer architectures. Although the transition to multicore platforms allows for the applications to run in parallel, their access to resources are no longer exclusive [1, 29]. In multicore platforms, the number of functions as increased, and it is expected to continue to grow in the future [2]. As the pressure to get smaller, lighter, and power-efficient have made it increasingly necessary that real-time embedded systems integrate different levels of criticality on a common hardware platform multicore platforms are being leverage to achieve this goal[1, 2]. This approach raises all kinds of processing delays caused by the interference generated from the shared system resources, for example, processors, memories, bus, I/O devices, and the creation of interference channels [30].

Interference can cause a variety of delays that could threaten the performance of an application in a way that is difficult to predict. This problem becomes relatively important if the performance is shared with applications with different levels of criticality where higher criticality partitions need to fulfill deadlines deterministically [2]. It is in the interest of providing isolation between the different levels of criticality to guarantee performance. Thus, critical applications should not be affected by the execution of non-critical applications or at least the interference between them should be contained.

However, as most hypervisors have been developed to virtualize the primary hardware resources [2], they still lack the proper management of resources that many guests may share, even if a dedicated core is assigned to each domain, memory accesses by these guests may interfere with one another.

One of the primary sources of interference is in the last level cache (LLC), where contention of lines can unpredictably be evicted, so memory access increases. With the rise of memory accesses, another problem that emerges when accessing the main memory is that DRAM controllers re-order access requests to maximize throughput, thus making it even harder to predict system behaviour. Thus, one of the most critical features needed in virtualization is the need for better isolation in the shared resources.

2.3.1 Cache Partitioning via Cache Coloring

The LLC was isolated using cache coloring [31], a well-established software-based approach for index-based cache partitioning. This approach uses the behavior of set-associative caches, which utilize a portion of the PA to choose which cache line to employ.

The LLC may be partitioned into domains executing on separate CPUs by simply designating non-overlapping sets of colors. Consequently, each domain will see a separate LLC partition with the illusion of disposing of a smaller amount of cache memory. It cannot experience evictions from the program

running in the other domains. Even if the system performance may suffer due to the decreased cache memory, each domain becomes more predictable and resistant to misbehavior attacks that originated in other domains.

Cache coloring serves three purposes, two of which are particularly important in the context of hypervisors [24]. First, it is practical because cache coloring may be efficiently implemented, does not require particular or unique hardware support, and is invisible to the application programmer. Second, hypervisors often control the allocation of memory regions for domains to virtualize their address spaces, resulting in a software architecture prone to including coloring techniques. Third, it enables the system designer to expose a reasonably primary configuration interface that is likewise transparent to the program running within a domain.

2.3.2 DRAM Interference

DRAM is another shared resource between cores. When running applications in a multicore chip concurrently, they compete for access to the main memory, which has limited bandwidth [29]. This limitation in the memory bandwidth, if not properly managed, can lead to harmful interference degrading the performance of the system and individual application. This scenario identified two main problems: spatial isolation and temporal isolation. Spatial isolation will be needed to be ensured by the hypervisor to separate memory space between guests. The two stages translations provided by the ARM virtualization are used to ensure spatial isolation.

Although last-level cache interference can be contained through different methods, like cache partitioning, it is still possible to interfere due to concurrent access to the DRAM main memory, mainly due to L2 cache miss. This problem can introduce delays to certain guests who need real-time guarantees. The problem occurs when one of the guest's applications behaves abnormally by making measureless accesses to the main memory, causing significant interference to other guests. Another way this problem can occur is when all the cores access the DRAM at the same time and make multiple accesses, causing delays to the other cores and creating dependencies between applications and different cores, as illustrated in Figure 2.12. These interferences can invalidate time executions requirements on tasks running on a core. In multicore systems, access to memory can vary depending on the access location and the DRAM controllers. There is a dependency between cores, since memory accessed by one core can influence the requests of another. The DRAM controller uses scheduling algorithms to re-order requests to maximize DRAM throughput. All these factors affect the temporal predictability of the main memory. This problem can be contained rough a memory bandwidth reservation mechanism. This mechanism limits the number of access to the main memory in a time window, and it is applied to each vCPUs. The memory bandwidth reservation provides each vCPU a budget in a time window. If this budget is consumed, the core is throttled. The mechanism is implemented by monitoring memory accesses using the PMU (see section 2.1.3) that collects statistics on the operations performed by the cores.

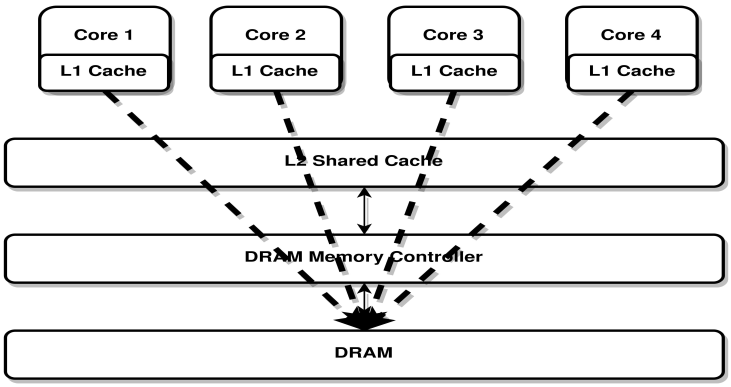


Figure 2.12: Every core accessing the DRAM at the same time.

Another problem that surges when talking about DRAM is that the DRAM consists of multiple resources called banks, being that these banks can be accessed in parallel, that can also mean that the performance of the system varies depending on how the data is stored in these banks and how these banks are shared among cores [32]. This problem has the best scenario and the worst scenario, as illustrated in Figure 2.13. For instance, the best scenario is where all the cores are accessing data in different banks, and the worst scenario is where all the cores are accessing the same bank, which would cause delays in accessing the memory. Although DRAM controllers are normally set to interleave the banks in order to improve bank-level parallelism [32], this causes a problem even though guests do not share the same memory space due to isolation, but at a given point, they may share banks. This is because OSes view the DRAM as a whole and not several banks when allocating memory. When DRAM controllers employ the interleaved bank strategy, this means that memory blocks at the size of memory pages are allocated to different banks [32], which means that any core can access any bank. This strategy causes problems when the application memory is allocated to different banks. It causes degradation to the system because the banks are shared between all the cores. For example, if two applications running in different cores access two different rows in the same bank, that means that the controller pre-charges a different row at each access performed to the bank. This latency accessing the bank can increase degradation in both applications, creating a dependency between applications in different cores [32]. An approach to this problem would be allocating memory pages in a selected DRAM bank for each application.

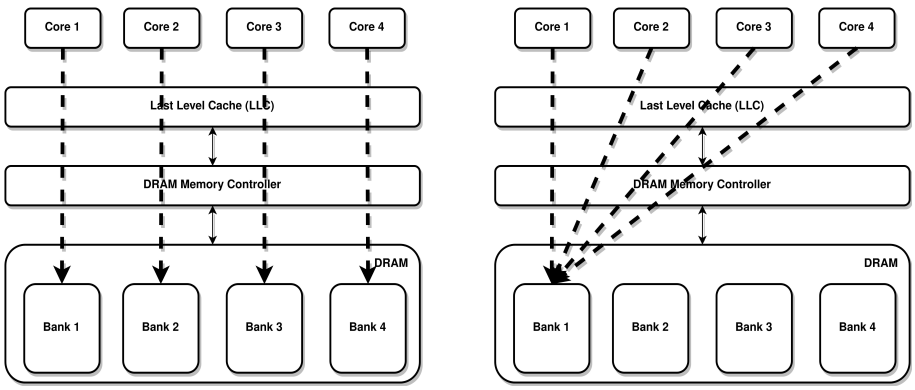


Figure 2.13: Best and worst case when accessing the memory.

2.4 Related Work

Regarding the memory bandwidth reservation for temporal isolation, systems employ their own implementation of the mechanism, depending on the desired use and the project characteristics. This section presents some essential implementations for this project, either for employing similar features or for being necessary for the overall project.

2.4.1 Memory Bandwidth Reservation implementation in XVisor hypervisor

Modica et al. [24], implements memory bandwidth reservation at the hypervisor level in an ARM processor, and the hypervisor is the XVISOR. It provides each domain a budget for memory access for each vCPU, replenishing the budget at each period. It uses ARM processors' PMU to track each vCPU memory access. The PMU provides certain events that can be counted, and the Modica implementation uses the "data memory access" event. This event tracks the number of accesses to the DRAM memory made by the vCPU, which manages its budget. It also uses another feature of the PMU that is the possibility to generate an interrupt when an event counter overflow happens. It uses these features to determine when the budget is depleted.

Looking at the integration of the mechanism with the vCPU scheduler of XVISOR, it was required to add a new state to the scheduler to integrate the mechanism. This new state was required to suspend the vCPU when it exhausted its memory budget and waits for a new recharge, and this state is a reaction to the event overflow. It is used a periodic timer to make budget recharges and initialize procedures. The Modica implementation has a state machine where it shows the integration between the memory bandwidth reservation with the XVISOR vCPU scheduler, as illustrated in Figure 2.14.

This state machine shows that upon creating the vCPU, it will transition to a state called RESET, where the domain control block is initialized with the memory reservation parameters. After so, the vCPU is ready to be recognized for execution, where it will move from the RESET state to the READY state. As stated before, a timer is set to make periodic recharges. When an interrupt comes, the timer needs to be restarted to implement the periodic behavior, and this timer is stopped when the vCPU is deactivated, thus needing to move back to the RESET state. Another function of the scheduler is that due to vCPU preemptions, the vCPU alternates between the READY and RUNNING state. When the vCPU is preempted, the current budget is saved until it transitions to a RUNNING state. When the PMU event interrupts overflows, the vCPU exhausted is budget, which means that the vCPU is suspended and changes from the RUNNING state to the RECHARGE. The vCPU will not be recognized for execution until the next timer interruption. Consequently, the PMU is deactivated. When the timer interrupt comes up, the vCPU will be recharged and recognized for execution, transitioning from RECHARGE to READY. To finish, when in the RUNNING state, the memory accesses made by the vCPU are counted by the PMU.

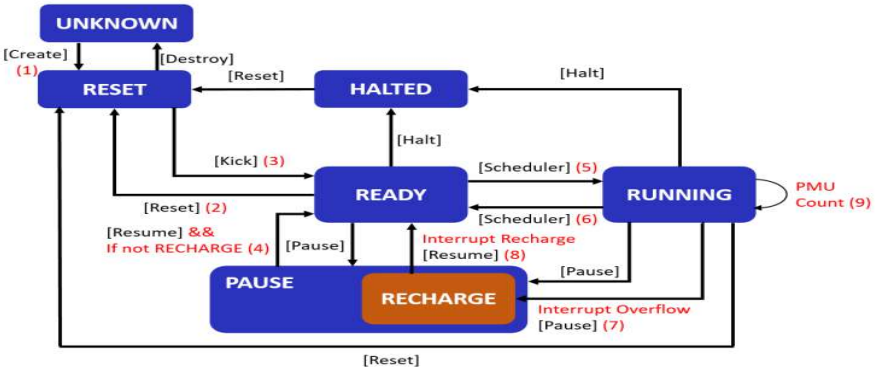


Figure 2.14: State Machine.

2.4.2 MemGuard

MemGuard it is an implementation of memory bandwidth reservation. It applies a regulator for each core to manage the bandwidth and a reclaim algorithm. MemGuard is employed at the OS level and has it applies a mechanism that allows the CPUs to reclaim budgets that other CPUs do not use. It can mainly support soft real-time systems. MemGuard bandwidth has two main modes: guaranteed and the best effort. The guaranteed bandwidth is the worst-case bandwidth of the DRAM memory, and the rest of the bandwidth is the best-case and cannot be assured by the system. This guaranteed bandwidth ensures that the system can provide a minimal performance to each core. Its goal is to provide performance and isolation while having maximum memory bandwidth utilization. The system architecture of the MemGuard is composed of a per-core regulator and a reclaim manager. The per-core regulator has the job of enforcing its core memory bandwidth usage. It uses hardware to count the memory access usage. When this memory usage exceeds a predefined threshold, it generates an interrupt, not exceeding the minimal memory bandwidth. Each regulator comes with a predictor. This predictor is based on memory usage history so that the core can donate its unnecessary budget. When a budget is donated, the cores can start reclaiming extra budget when they depleted their budget. Its reclaiming manager holds a global reservation for receiving and redistributing the budget [33].

In the mechanism itself of memory bandwidth reservation, MemGuard has two different applications. It has a global bandwidth that is applied through all the system. This is a minimum bandwidth, so it does not exceed the minimal DRAM service rate. Furthermore, the other is a per-core bandwidth. Each core is assigned a fraction of the minimum DRAM service rate. Each regulator has a period of one millisecond to reserve memory bandwidth, and the period is equal to the scheduler tick. It uses a Performance Monitoring Counter (PMC) from Intel to account the memory usage, and it uses its interrupt overflow. The PMC is programmed at each regulation period to generate the interrupt when the budget is exhausted. The regulator tells the scheduler to idle the core when the interrupt fires. Thus the core can no longer access memory until the next period comes. At the beginning of the next period, the core gets back all its tasks, and the budget is fully replaced [34].

Each core holds a bandwidth statically as a baseline for the memory bandwidth reclaimer to compare

to the prediction. If the core prediction is lower than the statically assigned budget, the current budget will be set with the predicted budget. Moreover, each core also holds an instant budget that is used to program the PMC. This instant budget can vary at each period based on the predictor and the usage of the core bandwidth. The MemGuard predictor uses an Exponentially Weighted Moving Average (EWMA) filter to estimate the budget for the current period. This filter uses the previous memory usage as an input. The reclaim manager holds the global shared budget where it receives a surplus of the budgets of each core and redistributes. Each core only communicates to the reclaim manager to reclaim its budget and donate to prevent complexity in the mechanism.

A problem that occurs with the reclaim manager is that it is based on a prediction which means that due to mispredictions, the core may never be able to use its original budget because he may have given too much of its budget, and the other cores may already have reclaimed the donated budget. The way that Memguard responds to this problem is to allow the core to continue to access the memory expecting that he only uses its static bandwidth until the next period. The next period verifies if it used the budget and if not, the predictor tries to compensate the core.

MemGuard also has spare memory bandwidth sharing where it tries to use the best-effort bandwidth. This is only applied when all the cores have depleted their assigned budget. The rest of the time left on the period is considered a spare time where all the cores can continue to execute to maximize memory throughput until the beginning of the next period. It only applies when all the cores have used all the budget because if one core has not yet used it, allowing other cores to continue to execute it would bring intensive memory contention, and the core could not be used the remaining budget.

2.5 Conclusions

The two implementations in Sections 2.4.2 and 2.4.1 have the same base mechanism but have different approaches at different levels. Some key points are necessary for implementing the memory bandwidth reservation on the Bao hypervisor. Although the MemGuard is an excellent fundamental approach to the mechanism, it is only applied to the OS level and has referred it has a problem of mispredictions, so the core may give too much of its budget and may never get it back.

The Modica implementation is the type of implementation required to be made in the Bao Hypervisor, besides the fact that Modica implementation is in an ARM platform as it will be the Bao implementation. However, one of the main differences between these two hypervisors (Bao and XVisor) is that the Bao hypervisor does not have the notion of CPU state since it is a static partitioning hypervisor. Furthermore, the Modica implementation is only focused on the core itself. In the Bao implementation, it would be desirable to have a memory bandwidth reservation for the guest and not for the system as a whole, regardless of the guest having one, two, or three cores. This characteristic is what sets apart the Bao implementation from these two. The Bao implementation will also require some manager as the MemGuard implementation for each guest to control every core of the guest.

The SMMU allows better isolation between guests in what concerns the DMAs devices memory accesses. The hypervisor makes sure that the device is assigned to a particular guest by allowing the device only to see the memory-mapped to the guest he is assigned. Although the SMMU does its job, a DMA device can still be used to interfere with other guests. This is due to memory accesses made by the DMA device that goes unchecked. It can still be used to make countless memory accesses and by doing so it causes significant interference to other guest by causing various L2 cache misses and delaying other guests requests, because they need to access the main memory to retrieve the data they required.

The two implementations in Sections 2.4.2 and 2.4.1, focus on memory usage of applications and do not take into consideration the problem described in the above paragraph, that modern I/O devices transfer data to the CPU by writing directly to the DRAM. Thus, not distributing the memory bandwidth properly to the CPUs and allowing unrestricted access to the main memory by DMA devices.

So the second half of this dissertation will be looking for interferences at the platform level. The focus will be on DMA devices interferences to the memory and analyse if memory bandwidth reservation is feasible on DMA devices. If this approach is not achievable, it will be required an analysis of possible solutions.

3. System Specification

3.1 Zynq UltraScale+ MPSoC

Due to the first or the second phase requirements, it was necessary to utilize the PMU and the CoreLink QoS-400 regulators for the respective phase. In order to integrate all requirements on board, the Zynq UltraScale+ MPSoC by Xilinx embeds it all. So the experiments have been performed on a ZCU104 board equipped with a Zynq UltraScale+ MPSoC by Xilinx. This board comes with an ARM Cortex-A53 processor, a mid-range, low-power processor that implements the ARMv8-A architecture that supports all exception levels, EL3 to EL0, and for both execution states AArch64 and AArch32. The Cortex-A53 processor has one to four cores, each with an L1 memory system and a single shared L2 cache [13].

The Ultrascale+ CPU design is based on the Armv8-A processor architecture, which offers four unique permission levels for program execution known as Exception Levels (EL). Each EL, ranging from EL3 to EL0, is often assigned to a specific software component, such as EL3 for secure platform monitors, EL2 for hypervisors, EL1 for operating systems, and EL0 for user applications.

In addition, its exception model defines exception levels, with EL0 having the lowest software execution privilege and unprivileged execution at EL0. The remaining ELs indicate higher software execution privilege. It is also worth noting that EL2 and EL3 offer processor virtualization and secure state execution, respectively [13].

3.1.1 MPSoC

An MPSoC provides a high-performance processing system comprising programmable logic, a specialized graphics core, high-performance multicore applications, and real-time processors that are all integrated into a single unit. This approach minimizes the necessary external hardware, substantially lowering development costs, which is vital in application domains that demand low-cost solutions, including research and education. The Zynq UltraScale+, an MPSoC made by Xilinx, has a specialized graphics core and programable logic (FPGA).

ZCU PS Interconnect

The ZCU interconnect is centered around the advanced eXtensible interface (AXI) point-to-point channels used in the interconnect, which is part of the processing system (PS), are exploited to connect system resources and communicate addresses, data, and response operations between master and slave clients.

Data may be stored and retrieved by cores, accelerators, and I/O devices by accessing a globally shared off-chip DDR memory. The fact that some I/O devices include DMA modules, allows them to independently access memory, i.e., without the need for CPU involvement. Devices that can produce a lot of I/O traffic, such as the Gigabit Ethernet (GEM) module, frequently use this capability. On these systems, it is well known that accesses to DDR memory are among the most common sources of contention, especially when memory-intensive tasks are taking place, which might possibly make contention delays worse.

Analyzing in detail Figure 3.1, by noticing the red rectangles, it is noticeable another possible source of interference, as a consequence of these two devices intersecting in the Translation Buffer Unit (TBU). Considering a scenario where the Low Power Domain (LPD) accelerator starts making considerable traffic by transferring a large amount of data. This type of transfer could lead to a lot of new TLB entries, generating a large amount of TLB misses to other devices that share the same TBU as the LPD, for instance, the GEM device.

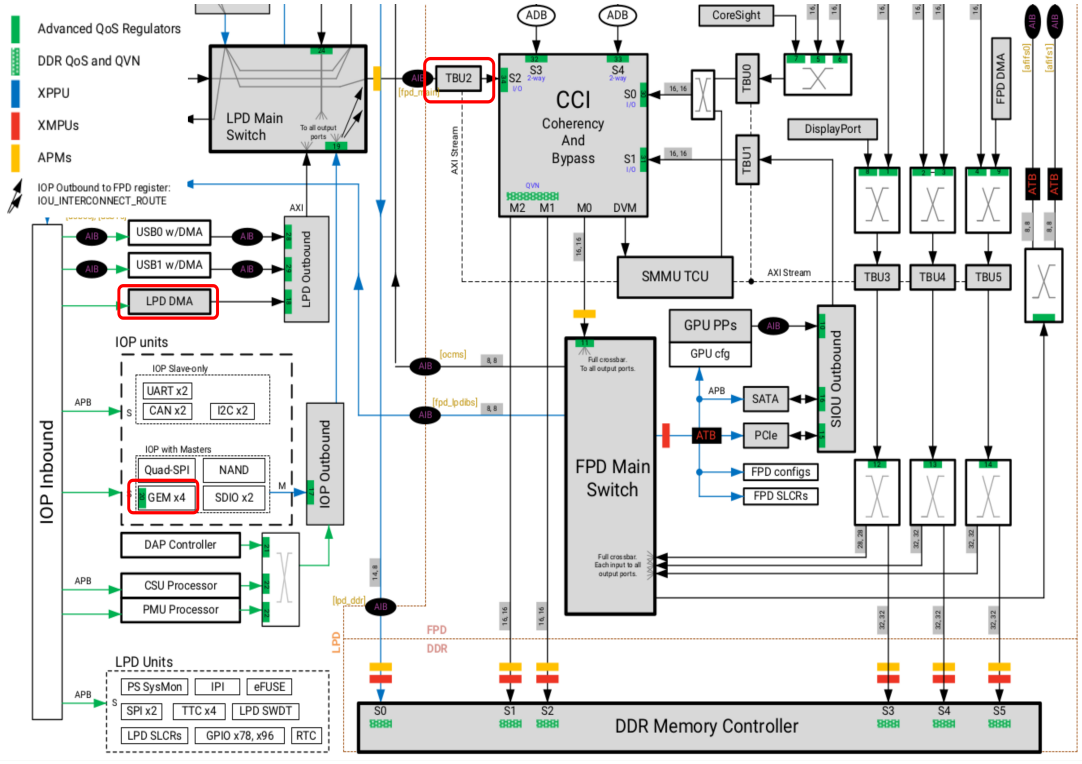


Figure 3.1: PS Interconnect.

The use of QoS regulators (see section 3.1.1) (green boxes in Figure 3.1) that incorporates functions to restrict the amount of bus traffic produced by the linked device or devices are positioned between I/O

devices and the rest of the bus as a means to regulate the amount of traffic generated by I/O devices reliably.

DMA

This section provides background information on the DMA technology, how it behaves, and how the different modes work to better understand this technology. External devices can access the main memory without the assistance of the CPU, thanks to DMA. The CPU often asks for a transfer from an external device. Then, the device may immediately transfer data from or to the main memory. The CPU is not stopped during the transmission and is free to do other tasks. The device notifies the CPU using an interrupt when the transfer is complete.

DMA Controller

Memory-to-memory and I/O-to-memory buffer transfers are supported by the general-purpose DMA controller. Low power domain (LPD) DMA and full power domain (FPD) DMA are two examples of a general-purpose DMA controller. I/O coherent access is possible for the LPD DMA. The FPD DMA transfers are never hardware coherent with the Cache Coherent Interconnect (CCI). The FPD and LPD DMA controllers are referred to as DMA in this dissertation. The only architectural differences between the 8-channel LPD and FPD DMA controllers are coherency, command buffer size, and data width.

- FPD DMA: 128-bit Advanced eXtensible Interface (AXI) data interface, 4 KB command buffer, non-coherent with CCI;
- LPD DMA: 64-bit Advanced eXtensible Interface (AXI) data interface, 2 KB command buffer, I/O coherent with CCI.

A general-purpose DMA, the Zynq UltraScale+ MPSoC DMA supports memory-to-memory, memory-to-I/O, I/O-to-memory, and I/O-to-I/O transfers. Every channel has its own independent on/off switch that may be used whenever desired. Simple register-based DMA and scatter-gather DMA are the two modes of operation that DMA provides. The source (SRC) and destination (DST) descriptors that the DMA implements are independent and can transport any size payload (up to 1 GB). The beginning and finish of a descriptor payload can be at any alignment.

Each DMA channel can be separately configured to one of the following modes:

- **Simple DMA Mode**

  Due to the fact that simple DMA mode only requires one command to transport data, it is also known as single-command mode. The DMA transfer parameters are defined in the control registers in basic DMA mode. These parameters are used by the DMA channel to transport data from the SRC to the DST side. In this single command mode, the DMA channel action is carried out once the transfer is complete. The next several steps are necessary for further transfers:

1. Set new additional transactional parameters to the control registers;
2. Enable the DMA channel.

There are simple DMA sub-modes. Only simple DMA mode supports the read-only and write-only modes. The following sub modes can be configured for each channel:

- In **read-only mode** where the data is not written anywhere by the DMA channel; it just receives it from the register-specified location. This sub-mode can be used to scrub the memory.
- In **write-only mode** where the DMA channel sends data to memory after reading preloaded information from the control registers. Data is not read from a memory location using the DMA channel. The registers that are utilized to write the DST locations are loaded by software with the source data.

- **Scatter Gather DMA Mode**

  The channel receives data from the address indicated in the SRC descriptor and writes to the location specified by the DST descriptor while operating in scatter-gather DMA mode. In order to accommodate descriptor storage in two forms, the DMA implements a hybrid descriptor.

  - Linear;
  - Linked-list;
  - Hybrid (multiple linear buffer descriptor arrays chained as a linked list).

  A hybrid descriptor can be used by software to dynamically transition between linear and linked-list mode. The DMA driver software can arrange descriptors in a contiguous array of Buffer Descriptors (BDs), a linked list of BDs, or a mixed mode where contiguous arrays of BDs can be chained together to generate a linked list of BD arrays thanks to the hybrid descriptor method.

  In order to identify the type of the current descriptor, each descriptor on the SRC and DST sides implements a bit descriptor-element type. This enables software to dynamically transition between a linear and link-list approach.

  - **Linear Descriptor Use Case:**

    In the use case mode for linear descriptors, BDs are kept in a linear array. The linear descriptor mode is displayed in the first block of Figure 3.2. This can be considered as one 4K page.

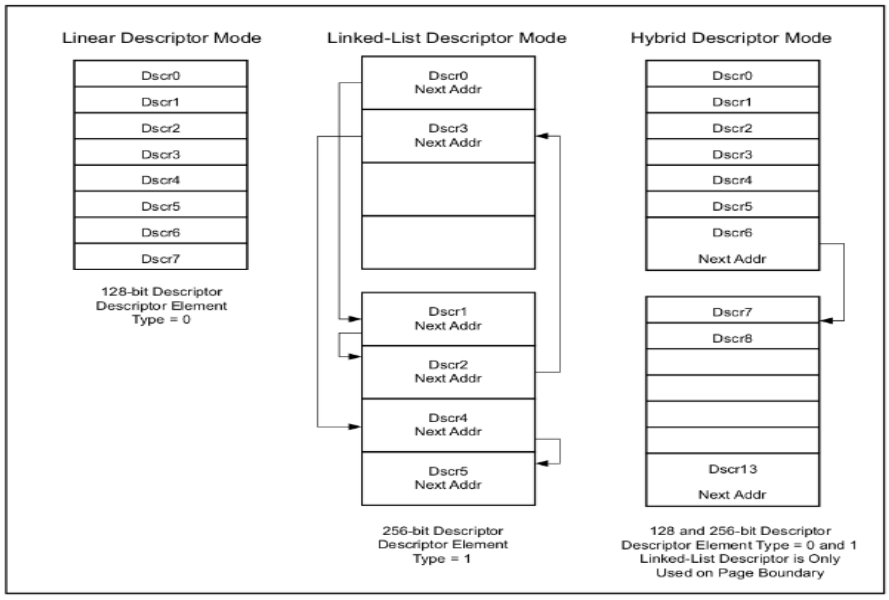


Figure 3.2: Scatter Gather DMA Mode: Hybrid Descriptor example.

- **Linked-list Descriptor Use Case:**

  A reference to the subsequent descriptor is provided by the final 128 bits of each 256-bit descriptor, the first 128 of which contain the descriptor's data. The descriptor can be found anywhere in the memory in this way.

ARM coreLink QoS-400 regulators

The ARM coreLink QoS-400 regulator incorporates functions to restrict the amount of bus traffic produced by the linked device or devices. Multiple regulators are provided in the system enabling the specification of various rules for various devices. In most situations, a specific regulator is included with each device. Transaction rate regulation, outstanding transaction regulation, and transaction latency regulation are the three operational modes that any regulator can use. In this dissertation, the interest is in the first mode. Each QoS-400 regulator is then set with three control parameters in the first working mode [35]:

- r_i (average rate): average allowed transactions per clock cycle;
- b_i (burstiness allowance): supplementary transactions budget;
- p_i (peak rate): maximum allowed transactions per clock cycle.

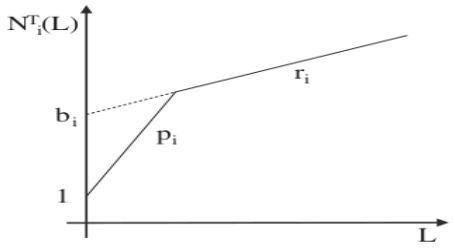


Figure 3.3: Bound on the maximum number of transactions the regulator will permit in a window of time of length L.

The QoS-400 regulator likewise permits the specification of two independent values, one for write address requests and the other for read address requests, for each of the three parameters r_i, b_i, and p_i.

3.2 Benchmarks

This section will describe the benchmarks utilized in this dissertation.

3.2.1 MiBench

Because the MiBench benchmark suite is supposed to reflect embedded programs used in the industry [36], the MiBench applications were employed as real-time applications. In the academic domain, this benchmark is a reference benchmark.

MiBench is a free, commercially representative embedded benchmark available to all researchers. MiBench has 35 benchmarks primarily written in C, divided into six subclasses, targeting a specific area of the embedded market. The six categories are Automotive and Industrial Control, Consumer Devices, Office Automation, Networking, Security, and Telecommunications. This dissertation only focuses on the Automotive subset because it includes three high memory-intensive benchmarks making it more susceptible to interference due to LLC and memory contention. In addition, this subset is a target domain of the Bao hypervisor, which has mixed-criticality systems.

3.2.2 IsolBench

To evaluate the system's performance, it was used the two micro-benchmarks provided by the Isolbench, a set of micro-benchmarks designed to quantify the quality of isolation of multicore systems. These two micro-benchmarks were the Latency and Bandwidth. Latency micro-benchmark is a pointer chasing synthetic benchmark that examines a single linked list that has been randomly shuffled. Latency can only generate one outstanding request at a time due to data dependency. Another synthetic benchmark is Bandwidth, which reads or writes a large array sequentially [37].

3.2.3 iPerf

iPerf is an open-source tool written in the C programming language. Moreover, it works in a client-server model and supports UDP and TCP. Therefore, we need to have two systems that both have the tool installed.

iPerf is a command line tool used to measure the maximum network throughput a server can support. This benchmark allows manipulating the window size of what will be transferred from the client to the server, as well as the transfer's bandwidth or even the transfer's interval time.

4. Memory Throttling

One of the problems in multicore platforms managed by a hypervisor is that many guests may share hardware, although they believe that they have a dedicated machine. Even if a dedicated core is assigned to each domain, memory access by these guests may generate contention at the memory bus. This type of problem has a significant impact on system performance, and in the case of real-time systems, in addition to performance, the system's operation is also affected because it depends on time constraints.

For these reasons, this dissertation is dedicated to analyzing and applying throttling techniques to manage the number of accesses to the DRAM. As described above, consider a scenario where exists two guests, Guest 1 and Guest 0. If Guest 1 starts making long accesses to the DRAM memory, this behavior results in a considerable interference to Guest 0.

When handling this unpredictability problem, a bandwidth reservation technique is implemented to limit the number of accesses a guest can make in each period. This technique is guest-agnostic. Nonetheless, with such implementation, a guarantee is given from the point of view of predictability.

4.1 Architecture Design

To create a correct implementation of the memory reservation mechanism, the system must monitor the memory accesses of all the CPUs assigned to every guest. This mechanism, at some point, will need to change the status of the CPU to IDLE until the next period arrives so that the status of the CPU changes back to EXECUTING. From a high-level analysis it will be required the following components:

- A period and a budget for each vCPU;
- A PMU counter to keep track of the number of memory accesses made by each guest;
- A timer to recharge the PMU counter and to wake up the vCPU if it was idle.

Figure 4.1 represents a minimal state machine of the actions required from the memory reservation system to properly function.

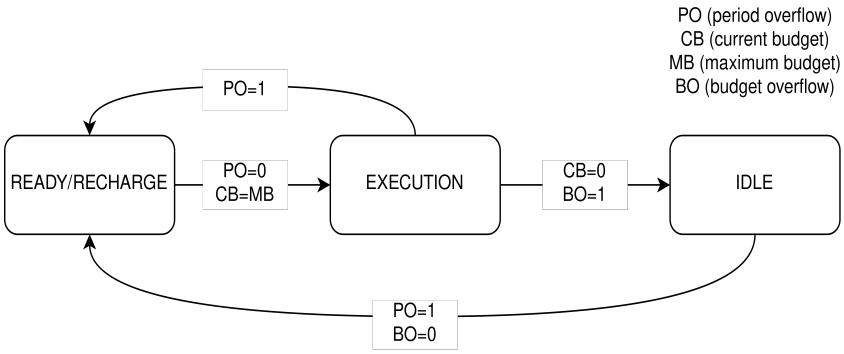


Figure 4.1: State machine for memory reservation.

Getting into more detail about how the memory reservation system works, first of all, each CPU is provided with its PMU, which makes this system applied to every CPU independently.

To transition from the EXECUTION state to the READY/RECHARGE state, it is necessary that the timer overflows, meaning that it is time for a new period. When in the READY/RECHARGE state, the main point of this new state is to recharge the budget given to the CPU and to clear the timer overflow flag. When this is done, it can go back to the EXECUTION state. In this state, the current budget is updated through the PMU counter that is set to count the memory accesses. If during the execution, the CPU ends their current budget, the CPU is suspended until the new period transitioning from the EXECUTION state to the IDLE state. At the beginning of the new period, the CPU is passed to the READY/RECHARGE state to recharge the current budget to the budget defined by the user and then passed to the EXECUTION state.

4.2 Implementation

This section will explain what features of the ARM PMU were used and how these have been exploited and integrated within Bao; it will also be explained what features of the generic timer were used and how they are applied to the implementation of the memory reservation system. The functions created in the hypervisor were named with a generic name so the implementation could be applied even for other architectures. For example, in the ARM architecture is called PMU, but for Intel is called PMC, and also because the name of the registers may change depending on the architecture, the way it must behave will not. Finally, it will be detailed how the memory reservation has been implemented in the Bao hypervisor.

4.2.1 ARM Generic Timer

The generic timer provides a standardized timer framework for ARM cores as the generic timer includes the system counter and not the cortex-A53 processor [13]. Figure 4.2 shows an example of the generic timer as a system timer. The system counter is an always-on device that provides a fixed frequency incrementing system count. This value is broadcast to all cores in the system, thus giving a common view of the passage of time. This passage of time is merely the passage of time and not the time or date.

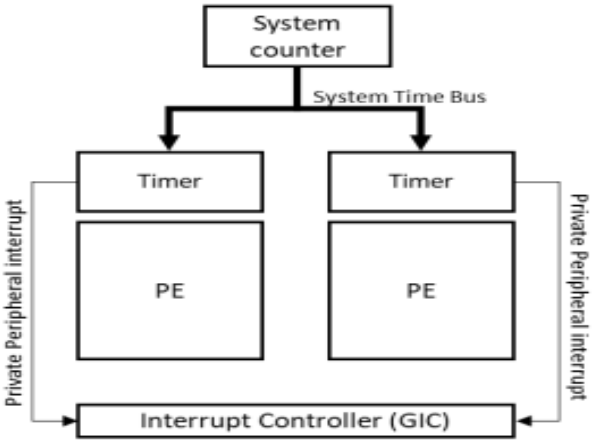


Figure 4.2: Illustration of a generic timer.

Each core has a set of comparator timers, as shown in Table 4.1 relative to ARM-v8 for Cortex-A53, which compare against the broadcast system count supplied by the system counter. These timers can be set up to produce interruptions or events at predetermined points in the future.

The following registers can program each timer after making the necessary changes to manipulate the correct timer. Table 4.1 shows every timer that can be manipulated.

The generic timer register are:

- **CNTFRQ_EL0** → Reports the frequency of the system count;
- **<timer>_CTL_EL<x>** → Counter-timer Control register;
- **<timer>_CVAL_EL<x>** → Counter-timer Comparator Value;
- **<timer>_TVAL_EL<x>** → Counter-timer Timer Value.

For instance *CNTHP_CTL_EL2*. For the implementation of this dissertation, it was used the "Non-secure EL2 physical timer". This timer is chosen because the mechanism's implementation is situated in the EL2 layer and thus utilizes a timer set for this layer.

Table 4.1: Cortex-A53 available timers.

Timer	Register prefix	EL<x>
EL1 physical timer	CNTP	EL0
EL1 virtual timer	CNTV	EL0
Non-secure EL2 physical timer	CNTHP	EL2
Non-secure EL2 virtual timer	CNTHV	EL2
EL3 physical timer	CNTPS	EL1
Secure EL2 physical timer	CNTHPS	EL2
Secure EL2 virtual timer	CNTHVS	EL2

Now that the registers are known, there are two ways to configure a timer. Configure by CVAL or by TVAL. By writing a value to the CVAL register, the timer only triggers when the count reaches or exceeds that value. When writing a value to the TVAL register, the processor reads the current system count internally, adds the written value, and then populates the CVAL register. In conclusion, if a timer event must be a trigger in X ticks of the clock, then it is better to program the TVAL register. If it wants an event when the system count reaches Y, it is better to program the CVAL register. To implement the memory reservation system is going to be manipulated the TVAL since it fits better with the goal of the mechanism, which is to generate a periodic interruption.

To generate an interrupt is necessary to configure the CTL register. This register as the following controls:

- ENABLE $\rightarrow$ Enables the timer;
- IMASK $\rightarrow$ Interrupt mask. Enables or disables interrupt generation;
- ISTATUS $\rightarrow$ When ENABLE is equal to one it reports whether the timer is firing.

To correctly generate an interrupt, the ENABLE control must be set to one, and the IMASK be set to zero, thus enabling when the timer triggers the signal to be passed to the interrupt controller that is usually the Generic Interrupt Controller (GIC).

The interrupt ID used for each timer to enable the interrupt in the GIC is in the Private Peripheral Interrupt (PPI) range, which means that these IDs are private to a specific core.

When an interrupt is generated, they behave in a level-sensitive manner, which means that the timer will continue to signal an interrupt until it is deasserted. This could be one of these situations:

- IMASK is set to one, which masks the interrupt;
- ENABLE is cleared to 0, which disables the timer;
- TVAL or CVAL is written, so that firing condition is no longer met;
- It is important that software clears the interrupt before deactivating the interrupt in the GIC. Otherwise the GIC will re-signal the same interrupt again.

When the Bao hypervisor is initialized, if the memory reservation system is activated to the target guest, then each CPU assigned to that guest is required to initialize the timer, as follows in the represented listing 4.1.

```
1 void timer_init(irq_handler_t handler, uint64_t period)
2 {
3     uint64_t frequency;
4     uint64_t count_value;
5
```

```
6   timer_define_irq_callback(handler);
7
8   //set timer counter
9   frequency = timer_get_system_frequency();
10  count_value = (period*frequency)/1000000;
11
12    timer_set_counter(count_value);
13
14  timer_enable();
15 }
```

Listing 4.1: Timer initialization code.

The following functions were developed to better manage the timer:

- `void timer_set_counter(uint64_t count)`
  Manipulates the TVAL register to set the clicks necessary to achieve the frequency wanted;
- `uint64_t timer_get_system_frequency()`
  Returns the frequency of the system count;
- `void timer_enable()`
  Allows to enable the timer;
- `void timer_disable()`
  Allows to disable the timer.

With this set of developed functions, the Bao hypervisor is able to enable and set the frequency it wants for the specific timer.

4.2.2 ARM PMU Support

A high-level overview of the PMU allows for a general description of its components. A common information known about the PMU, is that it is found in most modern CPUs and has the following components:

- One or more counters for counting events;
- Set of events that can be counted;
- Interrupt to signal a counter overflow.

The Cortex-A53 processor includes performance monitors that implement the Arm PMUv3 architecture [13]. These performance monitors allow for gathering various statistics on the operation of the processor and its memory system during runtime. These provide valuable system information that could prove helpful when assessing the performance and resource efficiency of the system and is often used to perform debug checks or performance measures.

The PMU makes 32 counters available at all privilege levels:

- 31 programmable event counters;
- A dedicated cycle counter.

But the Cortex-A53 only provides six programmable counters. Each counter can count any events available in the processor, and a counter dedicated to counting clock cycles. In Figure 4.3 shows a block diagram of the Cortex-A53 PMU structure.

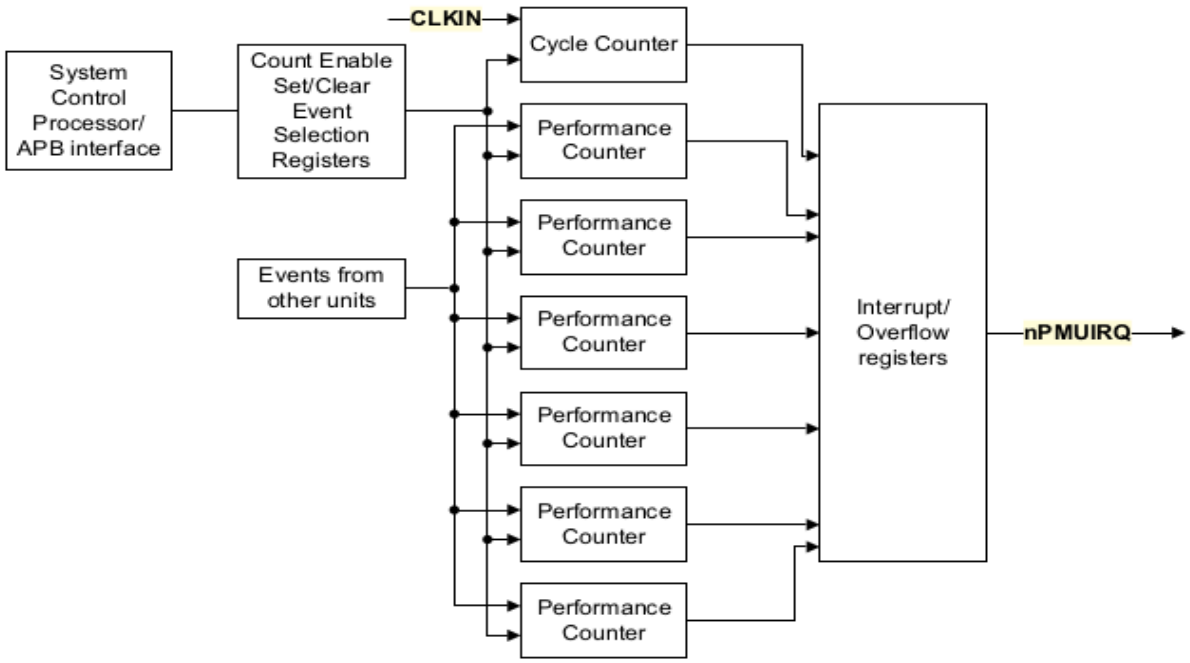


Figure 4.3: State machine for memory reservation from Cortex-A53 manual.

Each core has a dedicated PMU system with the above-described structure. The PMU allows for the record of architectural and microarchitectural events, and the registers can be programmable using the system registers or external APB interface. The PMU counters and their associated control registers are accessible in the AArch64 Execution state with MRS, and MSR instructions [13]. The central registers of the PMU are the following:

- **PMCR_EL0** → Performance Monitors Control Register;
- **MDCR_EL2** → Provides EL2 configuration options for self-hosted debug and the Performance Monitors Extension;
- **PMOVSCLR_EL0** → Performance Monitors Overflow Flag Status Register;
- **PMCNTENSET_EL0** → Performance Monitors Count Enable Set Register;

- **PMCNTENCLR_EL0** → Performance Monitors Count Enable Clear Register;
- **PMSELR_EL0** → Performance Monitors Event Counter Selection Register;
- **PMXEVCNTR_EL0** → Performance Monitors Selected Event Count Register;
- **PMXEVTYPER_EL0** → Performance Monitors Selected Event Type and Filter Register;
- **PMINTENSET_EL1** → Performance Monitors Interrupt Enable Set Register;
- **PMINTENCLR_EL1** → Performance Monitors Interrupt Enable Clear Register.

The procedure to enable a PMU counter is exemplified in Figure 4.4

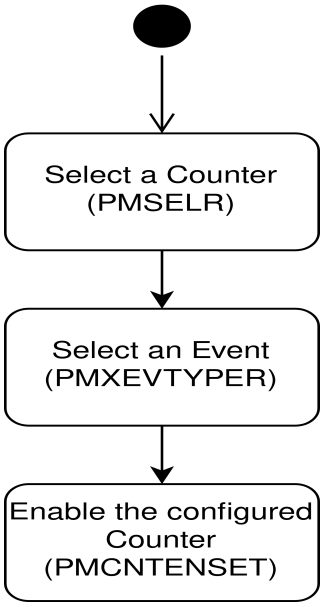


Figure 4.4: Simple PMU Setting.

Events Evaluation

The ARM PMU has many events that are possible to count, but this dissertation is restricted to the ones that the ARM Cortex-A53 implements. After looking at the events that the processor offers, a few stand out as useful to count memory activity in order to implement the memory throttling mechanism:

- **MEM_ACCESS, Data memory access** - The counter records all memory read and write operations performed by the core. Whether the access involves a Level 1 data or unified cache, a Level 2 data or unified cache, or both, the number is incremented [16];
- **BUS_ACCESS, Attributable Bus access** - The counter tracks memory read and write operations outside the core and its closely connected caches' boundaries. Data, instruction, unified cache refills, and write-backs are all examples of bus accesses [16];

- **L2D_CACHE_REFILL, Attributable Level 2 data cache refill** - The counter tracks the number of Attributable Memory read and write operations that the core performed that accessed at least the Level 2 data or unified cache and resulted in the filling of either the Level 1 data, instruction, or unified cache or the Level 2 data or unified cache. Any access that requires data to be retrieved from outside the cache, even if the data is not eventually allocated into the cache, is considered a refill[16].

Interrupt System

In the mechanism developed to perform the memory reservation, was taken the advantage given by PMU where each counter can be programmed to launch an interrupt when it overflows. It was taken advantage of this feature for all-purpose that allows to take actions when the interrupt arrives and not have the trouble of reading the counter's value from time to time to determine if the CPU exceeded the given budget.

It was required to create a system that configured the PMU. It allows configuring as many counters as the ARM Cortex-A53 processor allows, configuring any event, and configuring the interrupt system permitted by the PMU. When the Bao hypervisor is initialized, each CPU is required to configure the PMU, as exampled in listing 4.2 if the guest was configured to make use of the memory reservation system. Each core has its own interrupt id because the PMU interrupts are private to each core, and so each core has its dedicated handler for the interrupt. The handler will be explained in section 4.2.3.

```
1  void events_init(size_t counter_id, events_enum event, unsigned long budget
       , irq_handler_t handler)
2  {
3    events_set_evtyper(counter_id, event);
4
5    events_cntr_set(counter_id, budget);
6
7    events_cntr_set_irq_callback(handler, counter_id);
8
9    events_clear_cntr_ovs(counter_id);
10
11   events_interrupt_enable();
12
13   events_cntr_irq_enable(counter_id);
14
15   events_enable();
16
17   events_cntr_enable(counter_id);
18 }
```

Listing 4.2: Events initialization code.

To better manage the PMU it was required to develop functions that allow for better automation of the managing process. In this case, were implemented the following functions:

- `void events_cntr_enable(size_t counter)`
  Enables a specific counter of the PMU;
- `void events_cntr_disable(size_t counter)`
  Disable a specific counter of the PMU;
- `void events_cntr_set(size_t counter, unsigned long value)`
  Defining an overflow value for PMU counter;
- `void events_set_evtyper(size_t counter, size_t event)`
  Specifies which event to count identified by *event*;
- `void events_cntr_irq_enable(size_t counter)`
  This function allows to enable the counter overflow interrupt request;
- `void events_cntr_irq_disable(size_t counter)`
  Disable the counter overflow interrupt request;
- `void events_clear_cntr_ovs(size_t counter)`
  Clear the overflow status;

With this set of advanced functions, the Bao hypervisor can define any counter it desires and is able to define any event to count as long as the processor defines it. It is also able to define a threshold that, when it overflows, generates an interrupt.

4.2.3 Memory Reservation System implementation on Bao

In order to implement the memory reservation mechanism within the Bao, it will be required to add new parameters to the CPU structure provided by the hypervisor. Define the interrupt id and the timer id in the exiting definition of the platform in the hypervisor, in this case, the ZCU104 board. Furthermore, construct the necessary changes to the Bao algorithm to implement and integrate the mechanism. The parameters to add to the CPU structure are as follows:

- `budget:` - This variable will store the budget assigned to the CPU, that is, the number of allowed memory accesses in a period;
- `throttled:` Throttled is used to be aware if the CPU was throttled and to take action if it was;
- `counter_id:` The counter_id will store the id of the counter assigned;
- `period_us:` Will store the period designated to the CPU;

- `period_counts`: This variable will store the period in counts so that it will not be required to read through registers the frequency of the system every period to recharge the timer;

When initializing the CPU structure, it is required to set all values to zero and the throttled variable to false. When building the hypervisor, it is required to pass a config file where it is specified how many VMs are to initialize, their memory, devices, and many others. In that file, when activating the mechanism, it is a requisite to put in the desired VM the following line, *.safe_mem = 1*, this will ensure that the mechanism is initialized by setting up the timer as well as the PMU.

The hypervisor supports many platforms and in those platforms is the ARMv8-A Xilinx Zynq UltraScale+ MPSoC ZCU104, which defines relevant specifications for that platform. Moreover, specifications were added relevant to the mechanism in that context. These include defining the timer id and the interrupts id for the PMU.

When the mechanism is activated, the timer will be configured to raise an interrupt periodically using the value stored in the *period_count*. To this interrupt, it will be linked a callback handler that will be called when the interrupt arrives, as each core has its own timer, so each will have its own callback. The interrupt handler for the recharge event generated by the timer must perform numerous activities in order to recharge the CPU budget and, if necessary, alter the PMU.

Algorithm 1 Algorithm of the callback handler for the timer interrupt

1: Disabling the timer;
2: Restart the timer to count up to a new period, ensuring period events within the Bao;
3: Resetting the counter utilized by the mechanism;
4: Recharging the CPU budget;
5: Checks if the *throttled* variable is true or false;
6: **if** *true* **then**
    Enables the counter overflow interrupt that lets the system knows if the CPU has exceeded the budget assigned to him;
    Sets the variable *throttled* to false;
    end
7: Enables the PMU counter;
8: Starts again the timer with its new period.

This way is assured that each CPU has a specified amount of accesses set at the system's initialization at each period. Figure 4.5 shows a flow diagram of the timer handler flow.

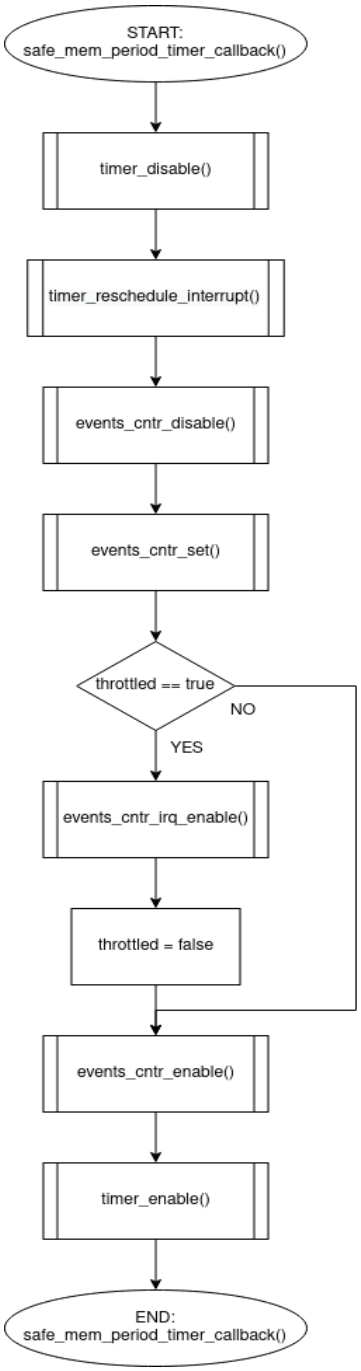


Figure 4.5: Timer Callback.

Like the timer, when the mechanism is activated, the PMU will be enabled to count an event set by the programmer, and will count the memory accesses made by the CPU during its execution. As stated in section 4.2.2, and like the timer, each core has its PMU. As the number of counters can vary from platform to platform, it was required to make the algorithm a bit more dynamic, setting a bitmap of the size of the number of the counter implemented for that platform. This can be achieved by reading the register *PMCR.N* that tells to whom is reading the register how many counters are implemented. With the

help of the bitmap is possible to allocate, to reserve a counter for the mechanism developed, and no other software in the hypervisor can manipulate this counter, even if the mechanism developed disables it.

Getting back to the initialization, it is also imperative to set an interrupt handler for the overflow interrupt to let the system know when the current budget is consumed, making it easier to know when it is necessary to suspend the execution of the CPU without the need for a polling system. An overflow interrupt can only occur when one of the counters goes into overflow. For this reason, it is necessary to start a counter from a calculated value so that the overflow occurs when the budget is depleted. This operations as described in section 4.2.2 are made by the *events_init* function, that in detail does:

Algorithm 2 Algorithm of the events initialization code

1: Sets the counter to count the event received in the parameter;
2: Sets the counter to a value calculated by the following formula: $UINT32\_MAX - budget$;
3: Sets the handler for the callback of the interrupt overflow;
4: Clears the overflow flag for the counter;
5: Enables the overflow interrupt;
6: Enables the general PMU and the counter.

The final description is about the PMU interrupt overflow, which means that the CPU ended their budget for that period, and it must be put in *IDLE* until the next period, thus changing the state of the CPU. These operations are executed in the handler of the overflow interrupt. It must do the following operations:

Algorithm 3 Algorithm of the counter overflow callback handler

1: Clears the counter overflow flag;
2: Disables the counter;
3: Disables the interrupt overflow;
4: Sets the *throttled* variable to True;
5: Enables the overflow interrupt;
6: Calls the function to put the CPU in *IDLE*.

Figure 4.6 shows the flow diagram of the PMU interrupt handler.

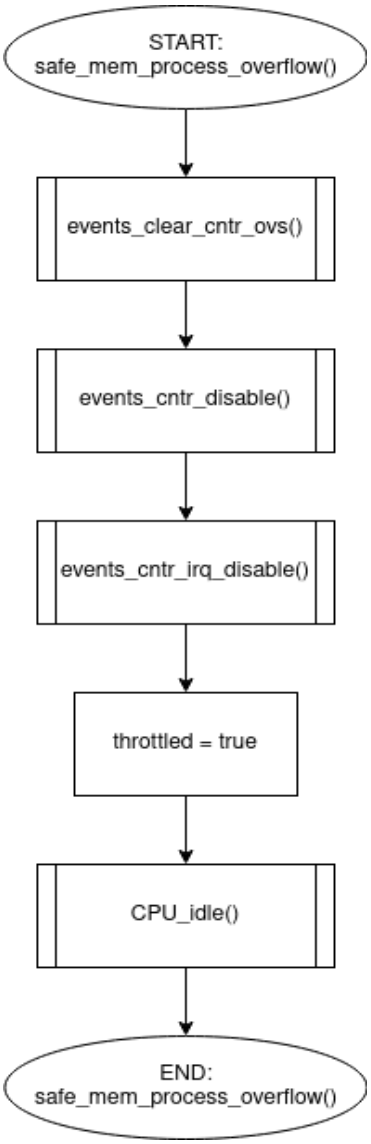


Figure 4.6: Overflow handler.

4.3 Evaluation and Results

In this section, the tests were conducted using a ZCU104 platform with an ARM Cortex-A53 processor. This section evaluates the results obtained by implementing the memory reservation mechanism.

4.3.1 Interference Application

This sub-subsection describes the interference generator used to oversee a malicious guest's effects that can happen to another guest. The guest used to generate interference is a bare-metal guest that operates on the EL1. Three CPUs were attributed to the bare-metal guest for maximum impact. At run time, the CPU will make consecutive reads from an array, and next will make consecutive writes to the

array. Each CPU has its own array. As the objective of this interference is to have the maximum impact on the other guest, to achieve this goal better, reads and writes are executed in such a way that each operation is performed on the cache line precisely after the previous operation.

By looking at Figure 2.12 is possible to see that when the CPU makes access to the memory, it flows a chain of command, first it searches if the information is in the L1 cache, if not it looks in the L2 cache, and so on. So when creating the interference, that was considered when defining the size of the arrays.

It were developed two scenarios, one to generate interference in the DRAM by delaying access to the memory and another to contain the interference only in the L2 cache shared by the CPUs. The size of the array accomplishes this. For example, the ZCU board L2 cache has a size of 1MB. If the objective is to contain the interference in the L2 cache with three arrays, the size of each array could not exceed one-third of 1MB. Furthermore, if the objective is to generate interference in the DRAM, the array size would need to exceed this value.

4.3.2 Memory Reservation Results

The tests realized were designed to compare the predictability and amount of interference by running one Linux guest that evaluates its performance by employing the widely-used MiBench Embedded Benchmark Suite and one bare-metal guest generating interference in the cache and in the DRAM. The tests were more focused on the automotive subset provided by the benchmark because it is one of the main application domains targeted by Bao [10].

Like [10], to better understand/view the problem, first, it was required to generate the same conditions and scenario as in the paper. To achieve this goal, a bare-metal guest with three cores was utilized. Each core accesses different regions of the memory so that with all these accesses, the cache would be complete, and new accesses to the memory would be required, generating an L2 cache miss. In this test it was employed four scenarios:

- **Solo** - Where only the Linux guest is running;
- **Solo-col** - Where only the Linux guest is running, but the coloring mechanism is being used;
- **Interf** - Where the Linux guest is running simultaneously with the bare-metal guest where this guest is generating interference for the Linux guest;
- **Interf-col** - Where the Linux and bare-metal guest are running, employing the coloring mechanism to fight the interference generated by the bare-metal guest.

Figure 4.7 shows the results of one hundred runs of the automotive subset. The results as performance normalized to the solo Linux guest execution case, where higher values represent poorer performance. These tests where the coloring mechanism is employed mean that only half of the L2 cache is available for each guest.

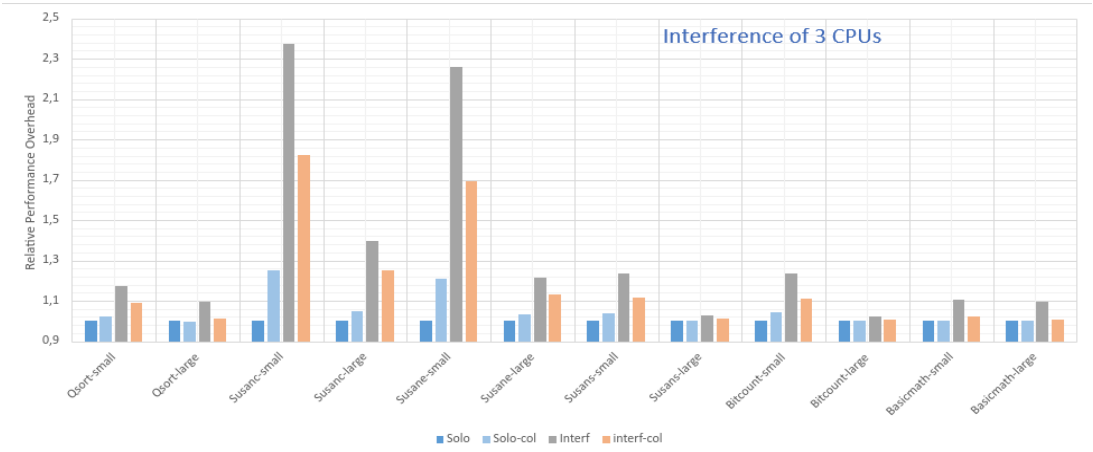


Figure 4.7: Performance overheads of Mibench automotive benchmark relative to a Linux guest execution.

Analyzing Figure 4.7, it is possible to observe the same tendency across the chart, to a greater or lesser extent. At first glance, it is noticeable that when the coloring mechanism is enabled, the performance overhead thrives rather more in comparison with the **Solo** test. Nevertheless, it is also noticeable that cache partitioning through coloring can significantly reduce interference.

Two more things are visible from this chart. One is that the performance overhead is more evident in the small data set variation of the benchmark compared to the large data set. Thus, coming to the conclusion that small input data set benchmarks will be more severely impacted by the cache miss penalty due to their shorter overall execution [10]. Second, since *basicmath* and *bitcount* use far less memory, these were substantially less affected by coloring and interference.

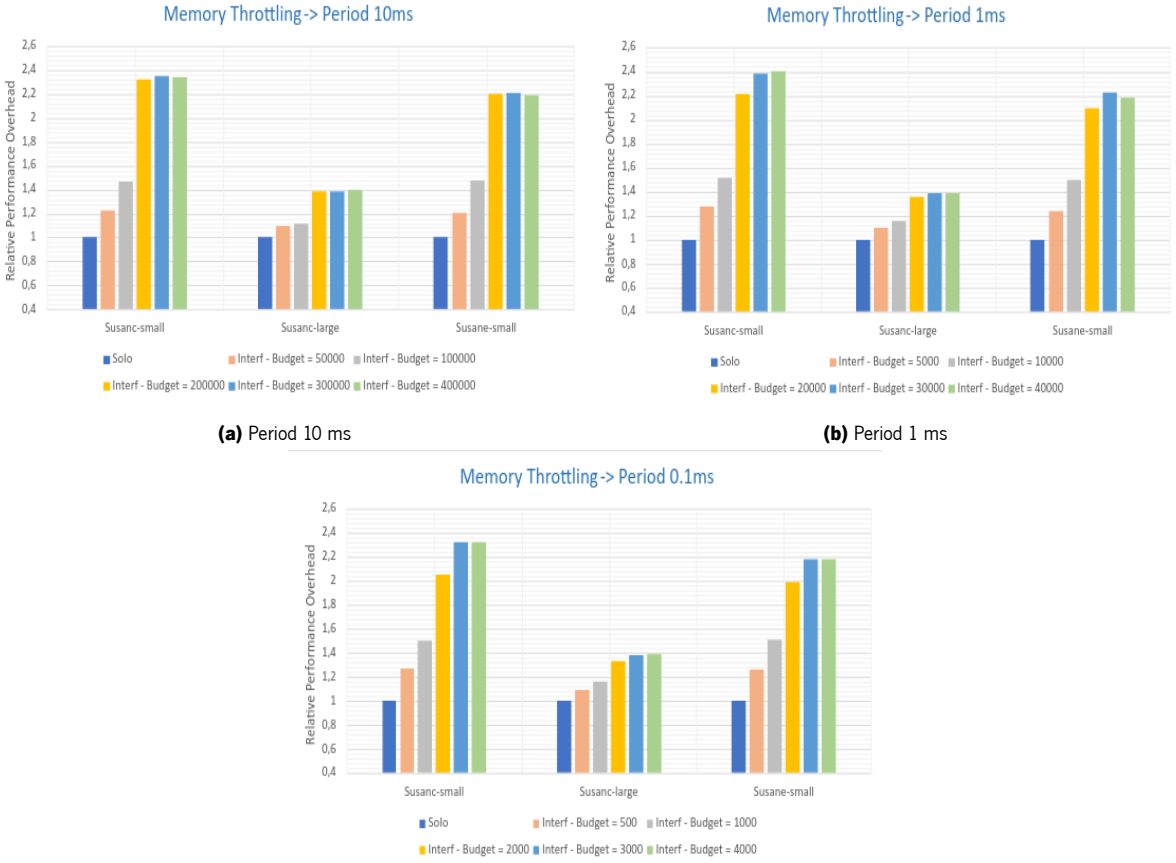
Now that the basic scenario is set, it is time to evaluate the benefits of using the mechanism implemented. Several experiments were conducted to evaluate the system's response. Furthermore, the tests were conducted with and without the mechanism to understand the results better. The system configurations utilized were the following ones:

- **Solo** - In this scenario, only the Linux guest is running;
- **Solo-Safe_mem** - In this configuration, the Linux guest is set with the throttling mechanism to see the impact that the mechanism has on the guest if enabled with it;
- **Interf** - In this configuration, the Linux guest is running simultaneously with the bare-metal guest where this guest is generating interference for the Linux guest;
- **Interf-col** - In this situation, the Linux and bare-metal guest are running, employing the coloring mechanism to fight the interference generated by the bare-metal guest.
- **Interf-Safe_mem** - In this situation, the Linux and bare-metal guest are running, but the bare-metal guest is enabling the throttling mechanism;

- **Interf-col-Safe_mem** - In this configuration, the Linux and bare-metal guest are running, both employing the cache partitioning mechanism, and the bare-metal guest is enabled with the throttling mechanism.

Experiments were conducted to see the implementation results for different sets of periods, specifically for 0.1 ms, 1 ms and 10 ms, and each period targeted the same bandwidth. For the upcoming tests that are not specified anywhere which event is being used, it is assumed that the test is evaluating the **BUS_ACCESS** event.

Memory Throttling Period Evaluation



(a) Period 10 ms

(b) Period 1 ms

(c) Period 0.1 ms

Figure 4.8: Performance Overheads of Mibench automotive benchmark for different budget values relative to a Linux guest Execution.

Figure 4.8 shows the results of the automotive Mibench subset. For each benchmark, the results are presented as performance normalized to the solo Linux guest execution. The tests run in these graphics use a configuration where the Linux guest is running simultaneously with the bare-metal guest where this guest is generating interference for the Linux, but the bare-metal guest is enabled with the throttling mechanism. For each benchmark, each column besides the Solo column represents a different budget for

the throttling mechanism. This test focuses on assessing, if transiting for a different period may change the mechanism's outcome. As the periods set to evaluate are in a logarithmic progression, the budget needs to be too. For instance, by setting the budget to 400 for the 0.1 ms period, when changing to a 1 ms or 10 ms period, the budget would need to change to 4000, and 40000 respectively.

One of the first conclusions that can be taken from these first graphics is that the mechanism is responding as expected, in view of the fact that by looking at either one of these three graphics is possible to view that the lower the budget set for the mechanism, the more interference it will take from the bare-metal guest, thus not generating as much interference as it would be if the mechanism were not enabled. Furthermore, as the budget keeps incrementing, more interference is being generated to the Linux guest.

Analyzing Figure 4.8 in more detail, it is also possible to notice that each benchmark from the three periods has approximately the same values. Thus, no matter the period chosen for the mechanism implemented, it will perform as expected.

Memory Throttling with Cache Coloring

This sub-subsection will be similar to the previous sub-subsection 4.3.2 but will be added the cache coloring to the mix. The point of this sub-subsection is to evaluate how the system would react when activating both mechanisms. This test assumes a period of 1 ms, and only the budget will be diverse.

Figure 4.9 shows the results of the automotive Mibench subset. The tests run in these graphics represent all configurations specified above in this section. By looking at these graphics is possible to see that both mechanisms independently are capable of diminishing interference coming from the bare-metal application. However, by examining the graphic where the budget is equal to 15000, the relative performance overhead when using the mechanism implemented exceeded the performance overhead when using the cache coloring mechanism. Revealing something already concluded: from the moment the bare-metal guest is given more budget, it can generate more interference to the Linux guest.

(a) Budget = 5000

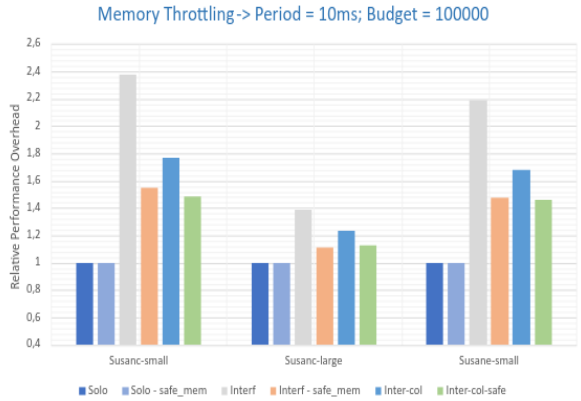
(b) Budget = 10000

(c) Budget = 15000

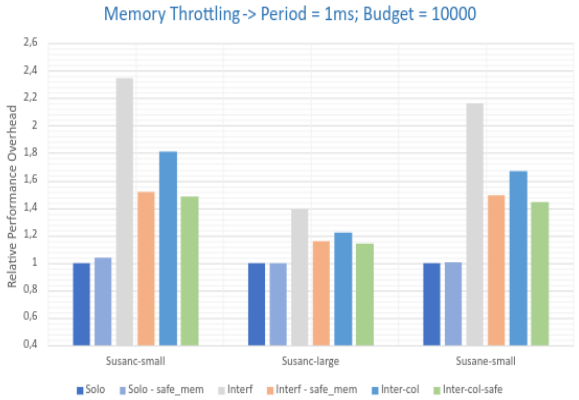
Figure 4.9: Performance Overheads of Mibench automotive benchmark for different budget values relative to a Linux guest Execution, analyzing the collaboration between the mechanism implemented and the existing cache coloring mechanism.

Another possible conclusion that can be taken from this figure is that by focusing on the two first graphics in the configuration where the two mechanisms are enabled. It is perceived that in the two first graphics, the relative performance overhead from this configuration and where only the memory throttling mechanism is enabled are relatively similar, so it is conjecturable that it is very similar to not having the coloring mechanism. Advancing to the third graphic, as the configuration where only the memory throttling mechanism is enabled exceeded the performance overhead of the scenario where only the coloring is enabled, this shows that for the budget of 15000, the memory throttling mechanism has a worst performance than the coloring mechanism. Now observing the scenario where both are enabled, it is noticeable that the relative performance overhead of this configuration is lower than when both are enabled but independently.

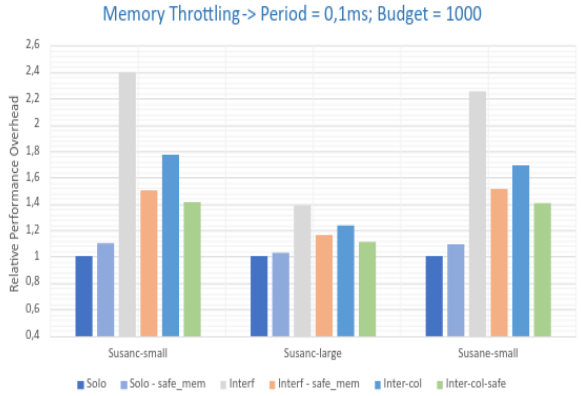
As this new test brought a new configuration where both mechanisms are enabled simultaneously, it is needed to evaluate how this configuration behaves when changing the period. Figure 4.10 shows the results of the automotive Mibench subset. The tests in these graphics represent different periods but with budgets in a logarithmic progression to ensure they have the same bandwidth.



(a) Period = 10 ms and Budget = 100000



(b) Period = 1 ms and Budget = 10000



(c) Period = 0.1 ms and Budget = 1000

Figure 4.10: Performance Overheads of Mibench automotive benchmark for different budget values relative to a Linux guest Execution, analyzing the collaboration between the mechanism implemented and the existing cache coloring mechanism in three different periods.

In Figure 4.10, the period with 1 ms was already tested. However, the other periods were not. Thus, observing the graphic where the period is 10 ms, it is noticeable, like in the 1 ms graphic, that the configuration where only the mechanism implemented is enabled, and where both mechanisms are enabled are relatively close. However, the configuration where both are enabled has a lower relative performance overhead. Furthermore, the same can be perceived in the 0.1 ms graphic.

Thus, in this sub-subsection can be concluded that the configuration where the memory reservation mechanism and the cache coloring mechanism are enabled is the best approach, considering the fact that when the budget of the memory reservation mechanism is too high, which means that if only this mechanism were enabled the interference generated by the bare-metal guest would have a more significant impact on the Linux guest. However, by enabling both mechanisms, the relative performance overhead will stall when it reaches the relative performance overhead of the configuration that only uses the coloring mechanism; this means that after a specific budget, it is comparable to only using the cache coloring mechanism.

Interference Only In The Cache

Until this point, the interference generated by the bare-metal guest would exceed the size of the L2 cache shared by the two guests, generating interference in the last level cache and the RAM. To see how the system would respond if the bare-metal guest interference would only make L2 cache contention and not exceed the cache size so that it would only interfere in the cache and not the RAM. For this to happen, as the guest generating the interference has three CPUs, each with its array to read and write memory, it was needed a change on the array's size to accommodate this new test. To ensure that the sum of interference generated by the three CPUs does not exceed the size of the L2 cache, the size of each array would need to be one-third of the total size of the L2 cache. Thus, the interference would be contained to the last level cache.

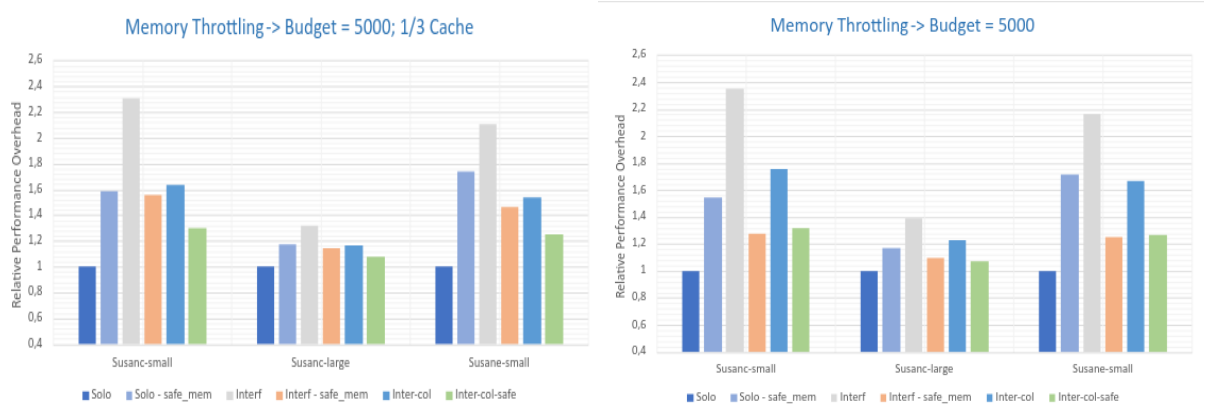


Figure 4.11: Performance Overheads of Mibench automotive benchmark relative to a Linux guest Execution.

Figure 4.11 shows two graphics for the period of 1 ms and using the BUS_ACCESS event, the left graphic exhibits the relative performance overhead when only interfering in the L2 cache, and the right one is a graphic of the previous sub-subsection 4.3.2. Thus, with these two graphics is possible to compare all the previous conclusions with this new context.

Analyzing the left graphic is interpreted that the configuration where only the mechanism implemented is activated has more performance overhead than the right graphic. Meaning that with this event, the interference generated by the bare-metal guest is not so well detected by the mechanism. However, it is also clear that the shift of the interference did not impact the performance overhead of the configuration where both mechanisms are activated.

As the configuration where only the memory reservation mechanism did not perform so well, a new test was conducted where the period remained the same, but the event was changed to the L2D_CACHE_REFILL, to grasp if anything would change for the better.

Although a different event could have been chosen, like L2D_CACHE_ACCESS event, the event chosen still needed to maintain a way to control what is going to access the RAM because the purpose of this

mechanism is to prevent memory contention and not cache contention, that why the cache coloring is in place.

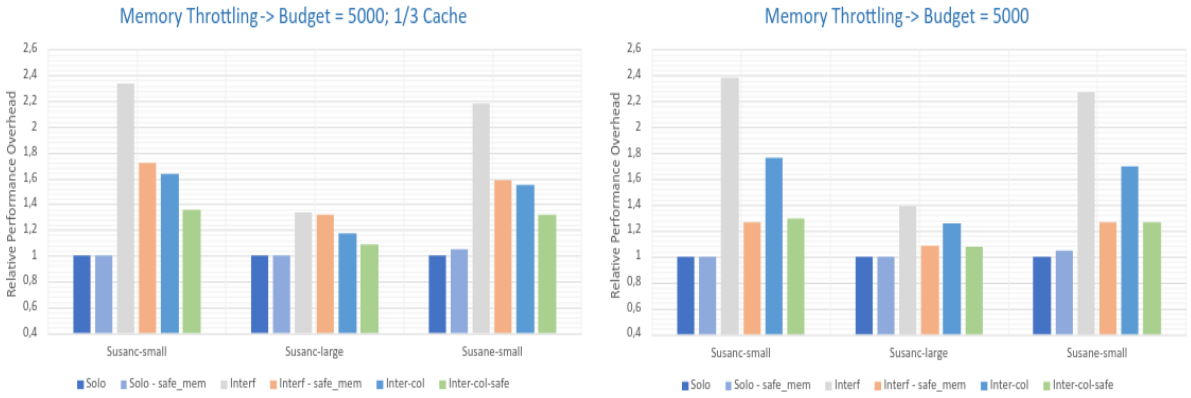


Figure 4.12: Performance Overheads of Mibench automotive benchmark relative to a Linux guest Execution.

Figure 4.12 shows two graphics for the period of 1 ms, and using the L2 cache refill event. The left graphic exhibits the relative performance overhead when only interfering in the L2 cache. The right graphic shows the performance overhead when the sum of the three arrays used to generate interference for the Linux guest exceeds the size of the L2 cache, generating interference in the DRAM.

Analyzing Figure 4.12 by comparing the two graphics and comparing them to Figure 4.11, it is noticeable that using this new event does not bring anything new to the table because having the interference contained entirely in the L2 cache, makes that the events of L2 cache refill will not have as many counts so that the mechanism can actuate.

Final Conclusion

To understand the difference between the two configurations better. A graphic was created to show for the period of 1 ms and different budgets, the relative gain of the performance overhead that these configurations have on the performance overhead of the configuration where only the Linux guest is running in parallel with the bare-metal guest without any mechanism enabled.

Figure 4.13 shows two lines, one is the scenario described above, and the other is the relative gain when only using the memory reservation mechanism.

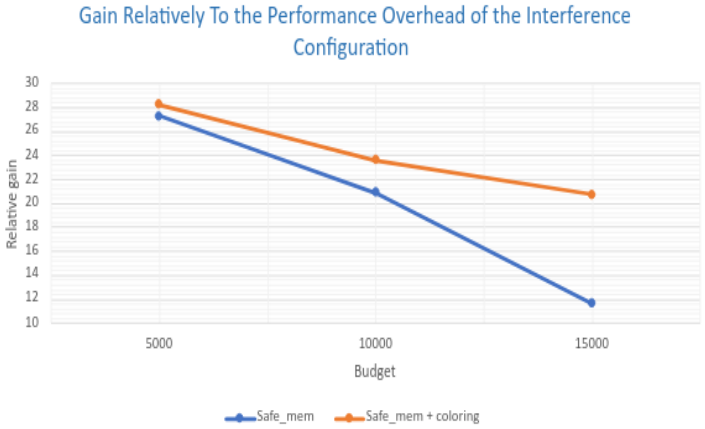


Figure 4.13: Gain of Mibench automotive benchmark relative to a Linux guest execution when running simultaneously with the bare-metal guest.

By analyzing Figure 4.13, it is evident that when both mechanisms are activated, the Linux execution performs better than when only using the memory reservation mechanism. For lower budgets, meaning that the mechanism actuates much more on the bare-metal guest, the deviation between the two lines is not very significant. Regardless, when the budget gets bigger, the configuration where both work together behaves much better. Thus, the two lines tend to grow apart as the budget increases.

In conclusion, when oscillating the period, the mechanism's performance is not affected. Suppose the interference restricts itself to the L2 cache. In that case, the mechanism does not have much power to act on it, even though the mechanism implemented was not developed thinking of L2 cache interference but having temporal isolation between the two guests. Overall, the performance of the two mechanisms working together is better than having them work individually.

5. Platform-Level Interference Study

This chapter is about the possible interference generated by DMA accelerators. This section will explain how the interference was generated using the DMAs for a better understanding of the results. Two types of interference will be generated, one more focused on DRAM and another more focused on TLB. For the first type of interference, the results will be observed whether using the LPD or the FPD. For the second type of interference, LPD channels will be used, as these tests concentrate on the channels that share the TBU. Finally, the results that start with the use of regulators to contain this type of interference will also be evaluated.

5.1 Implementation

- **DMA**

  It was also essential to develop a device driver for the actual device. The ZDMA driver from Xilinx was used for this. This device driver controls and configures the device. A single-device driver may manage all of the settings for the various channels.

  Designed to handle memory to memory and memory to IO buffer transfers, ZDMA is a general-purpose DMA. There are two instances of general-purpose ZDMA in ZynqMP. One is in the Full Power Domain (FPD), which is GDMA, and the other is in the Low Power Domain (LPD), which is ADMA.

  Each of the eight DMA channels in GMDA and ADMA can be designed to be secure or not secure. Each channel is separated into two functional sides: Source (Read) and Destination (Write). The following DMA modes allow for the independent programming of each DMA channel:

  1. Simple DMA;
  2. Scatter Gather DMA.

- **SMMU**

  When it comes to the SMMU performance monitoring extension, it was used on a stream ID basis to be able to filter the SMMU accesses to only the accesses made by the TBU shared by the LPD

channels and the GEM module. Another relevant fact is that a mask was applied only to count the number of accesses to the TLB made by the GEM module.

In this implementation of the SMMU performance monitoring extension, a timer was also used merely to periodically print the number of accesses made by the GEM module. The timer implementation is the same as the one performed in section 4.2.1.

- **QoS-400 regulators**

  The QoS-400 regulators cannot be accessed via the Xilinx standard software stack. Investigating this problem revealed that the regulators are reserved as privileged resources, prohibiting them from being utilized by either the Linux kernel or the EL2.

  The Linux kernel is designed to function in EL1, it is also presumed that the Arm Trusted Firmware (ATF) is operating in EL3. ATF is a standard implementation for securely handling processor-related low-level functions, including power management and coordinating the two processor worlds supported by Arm TrustZone technology, normal and secure-world.

  In order to access the regulator registers, it was required to manipulate these in the EL3 via the ATF.

  The ARM QoS-400 regulators on the Ultrascale+MPSoC are memory-mapped devices. Each QoS-400 regulator has the following registers for controlling the regulatory parameters discussed above:

  - qos_cntl(bits 1–0): bit 0 enables write regulation, whereas bit 1 enables read regulation;
  - ar_p(bits 31–24): the read peak rate is expressed as an 8-bit fraction of the number of transactions per cycle;
  - ar_b(bits 15–0): burstiness allowed for readings is indicated by an integer number;
  - ar_r(bits 31–20): the average read rate is expressed as a 12-bit fraction of the total number of transactiTons each cycle;
  - aw_p, aw_b, aw_r: the same bit intervals used for reads were used for writes to govern peak rate, burstiness, and average rate.

  The transaction rate was calculated using the value provided in the aw_r and ar_r registers, which hold the least significant bits of a 12-bit fractional portion of a number expressing the number of authorized transactions per clock cycle. For instance, a value of 0x800 (decimal 0.5) indicates that one transaction occurs every two cycles, and a value of 0x400 indicates that one transaction occurs every four cycles.

  The values utilized in this dissertation were based on [35], because it already does an extensive research with satisfactory results. In [35] the tests were performed with multiple values for the regulation of the device and concluded that for regulations above 0x100 the regulation does not

take effect. Thus, taking in consideration to not make this dissertation extensive it were only chosen two values for the regulation applied in this dissertation. The more relevant regulation values are:

- 0x20 regulation;
- 0x10 regulation.

5.2 Interference Application

This section is designed to explain how the interference is generated when using the DMA devices. As described in the subsection 3.1.1 the DMA controller has two types of modes. There are two types of interference to take into account. The first is interfering in the DRAM using a DMA device, which is perceived by the CPU. The other one is interfering in the TLB. This type of interference is mainly perceived by other DMA devices that share the same TBU that the DMA device used to generate interference.

The first type of interference is achieved by using the simple mode of the DMA device. The guest generating interference would only need to create two arrays, one for the source, where the guest wants to transfer from, and another for the destination, where the guest wants to transfer to. It would need to make the same transfer repeatedly to continuously generate the same interference. The array's size is what defines the amount of interference generated.

The second type of interference can still be achieved using the simple mode since this type of interference requires a significant amount of memory. If the simple mode is used, it will generate considerable interference to the CPU, which is not the purpose of this interference. It also would take more time for the device to make the transfer since it would need to transfer the entire page to trigger a TLB fill using the simple mode. As explained in section 2.1.2, the TLB is a cache to keep track of recently used transactions. So each entry of the TLB maps a single page table. Assuming that the page size is 4 kilobytes, and the TBU as two thousand entries, that would require that to generate significant interference, the device accesses double the number of page table entries. To only generate interference in the TLB it was used the Scatter Gather mode with the linked descriptor, as described in subsection 3.1.1. This interference will have an array with a predefined size, and the channel will only transfer sixteen bytes of each page the array occupies to another array. With this methodology, the interference in the DRAM will be contained and focus on generating TLB misses.

5.3 DRAM interference using DMA devices

In this case, the interference generated is more dedicated to the DRAM. Therefore the mode used for the DMAs is the simple mode since, in this type of mode, it is relatively simple to understand and use for the desired type of interference. The tests performed were designed to evaluate the amount of interference generated to a Linux guest that evaluates its performance using the MiBench Embedded

Benchmark suite. These tests focused more on the automotive subset MiBench offers because this is one of the leading applications that Bao focuses on. In this section, it will be evaluated the results obtained first when using the range of channels that the LPD offers, then using the channels that the FPD offers, and then applying both devices.

5.3.1 Interference using LPD Channels

Figure 5.1 presents the results obtained from the automotive MiBench subset. For the purpose of compacting the information and for better visualization of the results, it was selected the three more impacted benchmarks from the twelve benchmarks provided by the automotive subset. For each benchmark, the results are presented as relative performance overhead to the Linux guest execution. The results presented here use a configuration where the Linux guest runs simultaneously with a bare-metal guest, which is generating interference using the DMA devices. For each benchmark, the column beyond solo represents the number of channels used to generate interference. This test is focused on verifying if it is possible to generate interference to another guest using DMA devices. Different amounts of the channels were used to assess the impact they would have by varying this value.

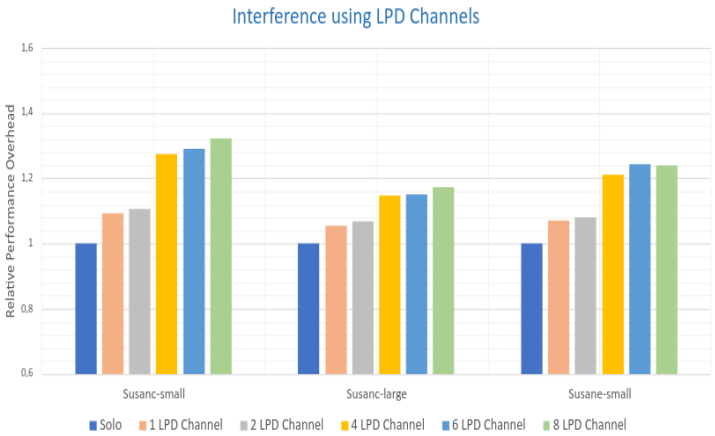


Figure 5.1: Performance Overheads of Mibench automotive benchmark when generating interference using LPD channels relative to a Linux guest execution.

Analyzing Figure 5.1 is conclusive that the main objective of this test is fulfilled, which is to assess whether using DMAs accelerators could be used to generate interference for other guests. By looking at this figure is observable and expected, that as the number of channels for interference increases, the greater the interference generated to the other guest will be. This pattern is repeated for all three benchmarks presented in Figure 5.1.

5.3.2 Interference using FPD Channels

As in the previous subsection, the results are presented in this subsection as relative performance overhead to the Linux guest execution. Due to the prior test's success, where it was concluded that DMA

devices could generate a significant amount of interference to another guest. Running the same test but changing to a different type of DMA device would be meaningless because the conclusion was already taken. However, this test is focused on evaluating if the FPD device generates as much interference as the LPD device.

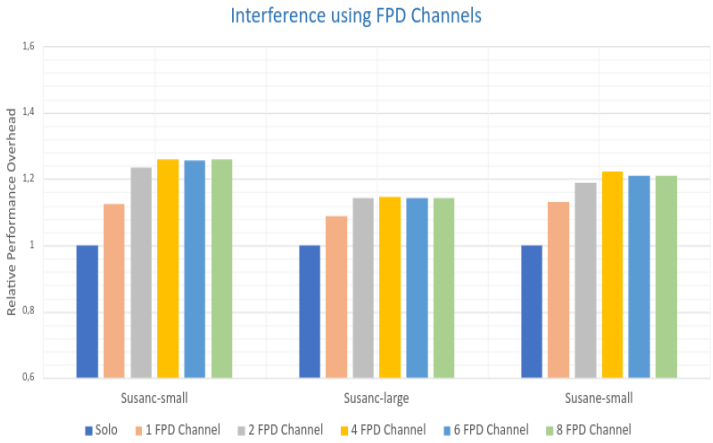


Figure 5.2: Performance Overheads of Mibench automotive benchmark when generating interference using FPD channels relative to a Linux guest execution.

Analyzing Figure 5.2 is concluded that when using the FPD device to generate interference, it fulfills its purpose. Another conclusion from this graphic is that after a certain number of channels, the interference generated becomes stable. In contrast, the interference generated by the LPD device keeps growing until the total number of channels available per device. Another conclusion extracted from the comparison of the two Figures 5.1 and 5.2, is that when using the FPD device, it does not generate as much interference as the LPD device. The slight performance overhead difference could be due to the fact that the LPD device is cache coherent, which means that this extra overhead comes from interfering in the cache level.

5.3.3 Interference using LPD and FPD Channels

Figure 5.3 presents the results obtained from the automotive MiBench subset. The results presented here use a configuration where the Linux guest runs simultaneously with a bare-metal guest, which is generating interference using the LPD and FPD devices. For each benchmark, the column beyond solo represents the number of channels utilized by the LPD and FPD to generate interference. This test is focused on verifying how much impact could be caused to another guest by combining the two devices. Different amounts of channels were used to assess the impact they would have by varying this value.

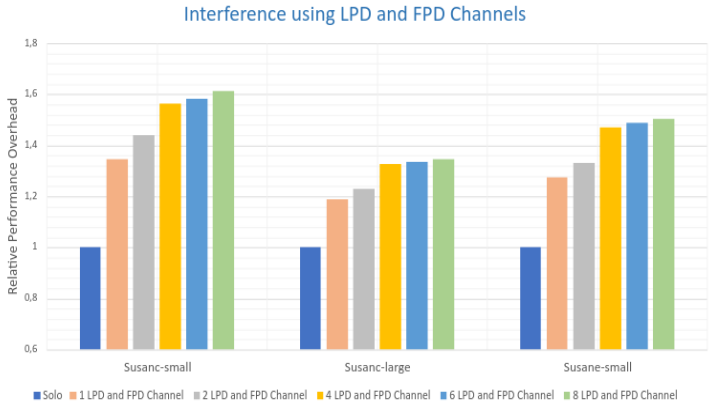


Figure 5.3: Performance Overheads of Mibench automotive benchmark when generating interference using LPD and FPD channels relative to a Linux guest execution.

By examining Figure 5.3 it can be concluded that merging these interferences, generated by these two devices makes the interference effect on the Linux guest much more significant than the two individually.

5.4 TLB interference using DMA devices

This section will explore the possibility of generating interference not in the DRAM but in the TLB. Systems perform better thanks to the TLB, which caches the virtual page to physical frame mapping. For real-time systems, TLBs are a source of unpredictable behavior. This unpredictability can result in TLB misses with a cost of up to thousands of cycles [38], worst-case execution time, and delays.

As it was explained in section 3.1.1, the fact that the GEM device and the LPD device share the same TBU could lead to contention for TLB entries. This could happen if one of these devices starts randomly making a significant amount of memory accesses.

To test this type of interference it were required a few components. The SMMU's Performance Monitoring Extension was then used to assess the number of misses that exist for a given device. In these tests, the device in which the number of misses was evaluated was the GEM. The choice of which device should be evaluated was made purely because the LPD device allows greater control over it, making it easier to generate maximum interference in another device.

It has been explained how the interference will be generated in section 5.2, and it has already been explained how and what will be evaluated. The only thing left to explain is how the GEM device will be used to witness how many misses this device has when it is handled and simultaneously suffers interference. A quick and easy way to manipulate the GEM device is to use *iPerf*. *iPerf* is a command line tool used to measure the maximum network throughput a server can support. This benchmark allows manipulating the window size of what will be transferred from the client to the server, as well as the transfer's bandwidth or even the transfer's interval time.

To better measure and understand how the interference impacts the GEM device, the number of TLB misses that this device has, was tested for different window sizes of the *iPerf* benchmark. These tests

were conducted for two modes of the LPD device. The first tests were conducted using the simple mode to evaluate how the number of misses behaves when accessing a large number of DRAM data. The second testes were using the scatter-gather mode with the linked descriptor. These second tests focus more on impacting the TLB, so behaviorally speaking, the second tests should generate a lot more TLB misses for the GEM device. In this case study, the GEM device is the client, meaning that the Linux guest manipulating the GEM device is sending packages to the server.

5.4.1 First TLB test

In this first TLB test, as previously described, it is used the simple mode of the LPD device. As the SMMU Performance Monitoring Extension from the ZCU104 board does not provide a TLB miss event, then to evaluate the behavior of TLB misses, it was used the *write* event. This is very close to the *miss* event because a write to the TLB will follow when a miss occurs. For the TLB tests realized is assumed the use of 4K page tables.

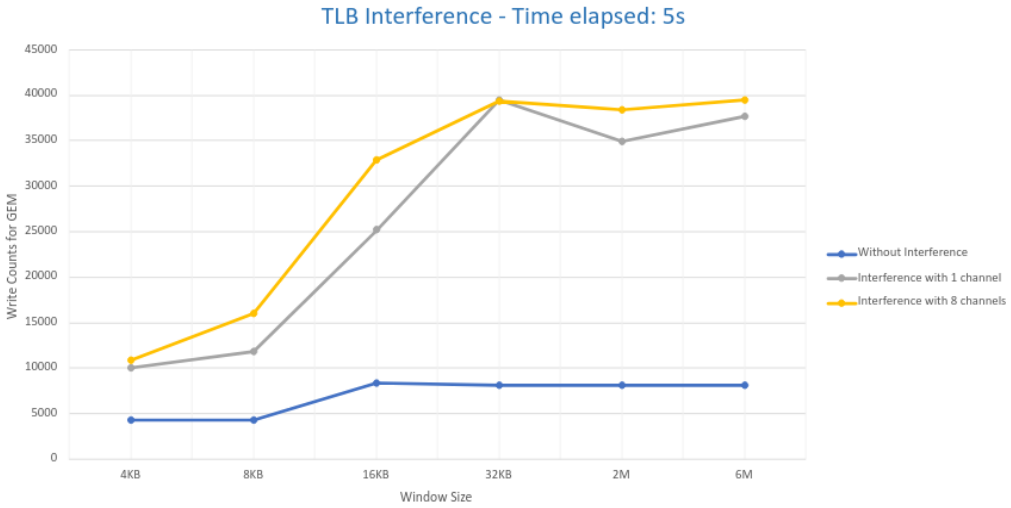


Figure 5.4: *Write* event count for GEM device, for different windows sizes, with and without interference.

Figure 5.4 shows the results of the *write* event count for the GEM device. The tests run in this graphic represent different configurations. These configurations are:

- The first scenario: Where the GEM device is running without any interference to achieve a base guideline;
- The second scenario: Where the GEM device runs in parallel with a bare-metal guest in charge of generating interference to the GEM device with one LPD channel;
- The third scenario: The same as the second scenario, but the number of channels has changed to eight.

Analyzing Figure 5.4 it is possible to observe that when the interference is generated to the GEM device, the number of *writes* skyrockets. This reveals that if some malicious intent targets a DMA device, it is possible to significantly impact the number of misses for the desired device. To fight this type of interference, the ZCU104 board provides the QoS Regulators, described in section 3.1.1, to restrict the amount of bus traffic produced by the linked device or devices. With the use of these regulators, it is possible to change the transaction rate.

Regulation at 0x20

At this point, it has already been observed that if any DMA device makes long and continuous accesses to the DRAM memory, this will generate interference in terms of memory access times. It was also observed that this interference caused by the LPD device channels generates interference in the TLB. In order to try to regulate and examine the system's behavior, the QoS-400 Regulators available for the ZCU104 board were tested. The same scenarios of interference were used to study the system response to the use of regulators, but now activating these regulators. For these tests, the benchmarks of the automotive suite of miBench benchmark were rerun, and also the test where the number of misses the GEM device has for the different scenarios.

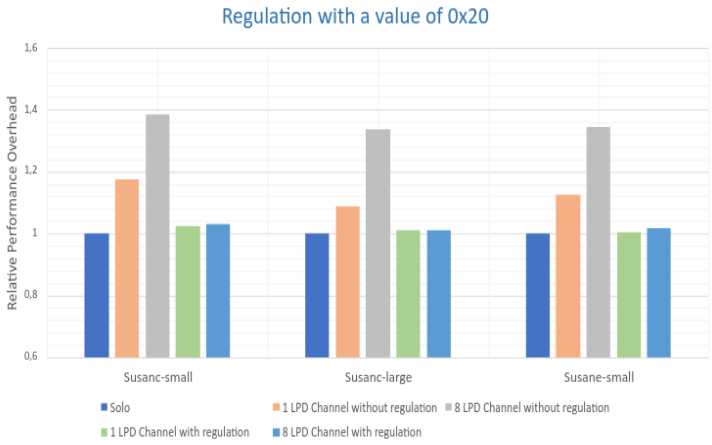


Figure 5.5: Performance Overheads of Mibench automotive benchmark when generating interference using LPD channels relative to a Linux guest execution with and without Regulation.

Figure 5.5 presents the results obtained from the automotive MiBench subset. In this test, two more scenarios are presented where it is applied regulation to the LPD device. These tests were conducted only using one channel and eight channels for the sake of space. Analyzing Figure 5.5 is concluded that when using the regulation, the performance overhead due to interference diminishes. It is still observable that even with the regulation, not all the interference went away. However, it is still a nice improvement. It is also a bit visible that the interference with eight channels has a little more performance overhead.

It is now evaluated the system's behavior regarding the interference in the TLB, but with the use of regulation in this device. It was performed the same test that Figure 5.4 shows but now with the use of regulators. As seen in the previous MiBench test, regulators also prevent the rise of misses generated by

the interference from the use of LPD device channels. However, for regulation of 0x20, it is still not enough to lower the number of misses to what is an interference-free scenario.

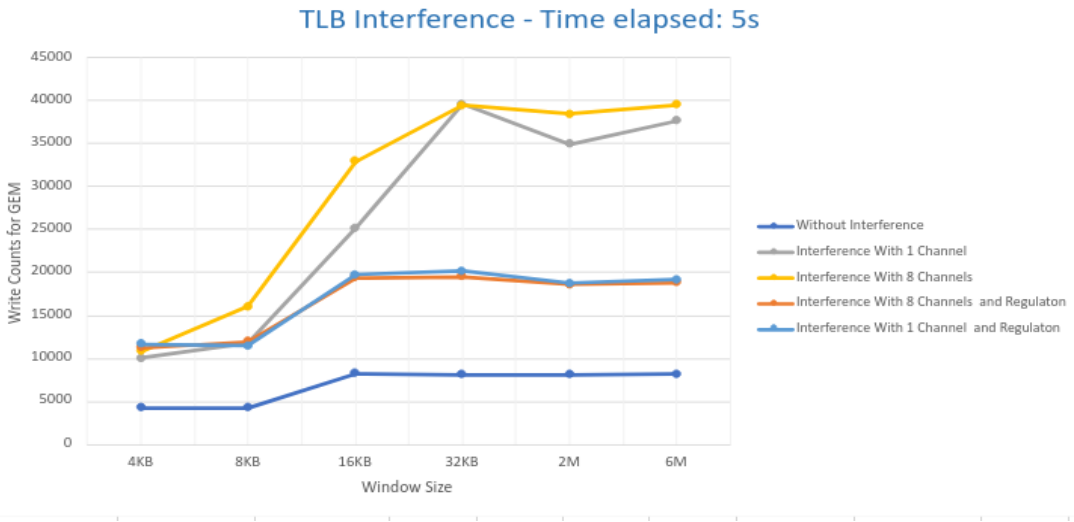


Figure 5.6: *Write* event count for GEM device, comparing the number of writes to the TLB when using regulation and not using the regulation.

Analyzing Figure 5.6, it is evident that for small window sizes, the effect of regulation is not felt as much because, for smaller window sizes, there will also be fewer accesses to memory. Therefore, it will not generate as many misses. Studying the figure, it is notable that from a window size of 16 kilobytes, the number of misses stabilizes to a value of twenty thousand misses, so the effect of regulation is not felt for smaller window sizes because for these windows, the number of misses does not exceed the twenty thousand.

Regulation at 0x10

For comparison purposes, the same tests were then performed as before, but for a setting of 0x10. Figure 5.7 presents the results obtained from the automotive MiBench subset. As the previous tests conducted for the 0x20 regulation, it was also added two more scenarios where it is employed the use of regulation to the LPD device.

Examining Figure 5.7 as it was assumed that since the test for a 0x20 regulation was successful, so would this test. So the only purpose that was kept in mind for performing this test was to compare the results of this test with the results obtained with a 0x20 regulation, Figure 5.5.

As is evident, the use of regulation positively affects Linux performance, as is well demonstrated in the two scenarios added to the test. Comparing Figures 5.5 and 5.7, a slight improvement in the overhead performance of all three benchmarks is vaguely noticeable.

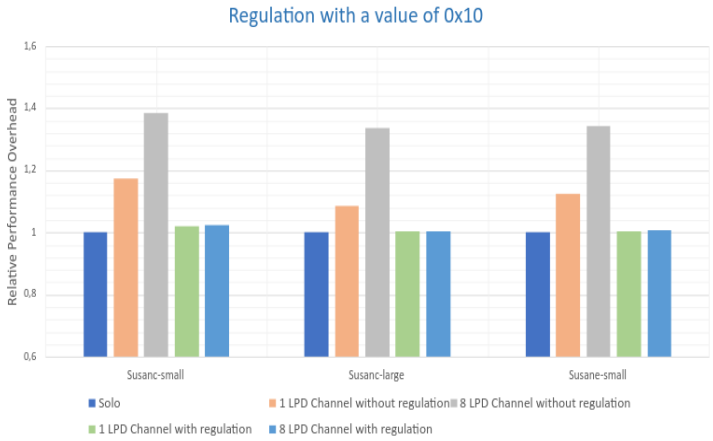


Figure 5.7: Performance Overheads of Mibench automotive benchmark when generating interference using LPD channels relative to a Linux guest execution with and without Regulation.

Analyzing Figure 5.8, the number of misses compared to the figure for a 0x20 regulation is much smaller. Now for smaller window sizes, we can see the effect of the regulation because, for a 0x10 regulation, the number of misses stabilizes at a value of ten thousand. As seen for smaller window sizes, the number of misses can vary between ten thousand and fifteen thousand, so the regulation limits this value to ten thousand misses. The lower the regulation value, the closer it gets to a non-interference scenario because the device takes many more machine cycles to complete a transfer, giving more space to other devices to perform their operations.

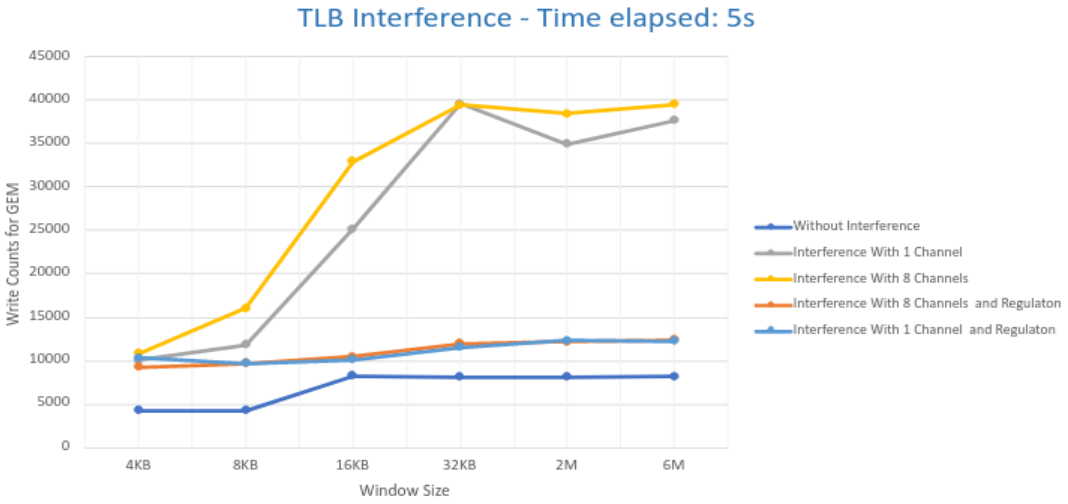


Figure 5.8: Performance evaluation for the GEM device by counting the number of TLB writes for comparison when applying regulation.

5.4.2 Second TLB test

Now that the first TLB interference tests are dissected, where the mode of operation of the LPD device, used to evaluate interference to DRAM, has been maintained, however, the focus of the evaluation was

shifted to the TLB. It is then time to move on to the next mode of interference that is more dedicated to generating a more significant impact on the number of TLB misses. So to this end, the simple mode of the LPD device channels was then switched to scatter-gather mode using the linked descriptor. The test described in this subsection was performed in the same way as in subsection 5.4.1, where the write event provided by the SMMU Performance Monitoring Extension was then used since it does not provide the miss event for TLB.

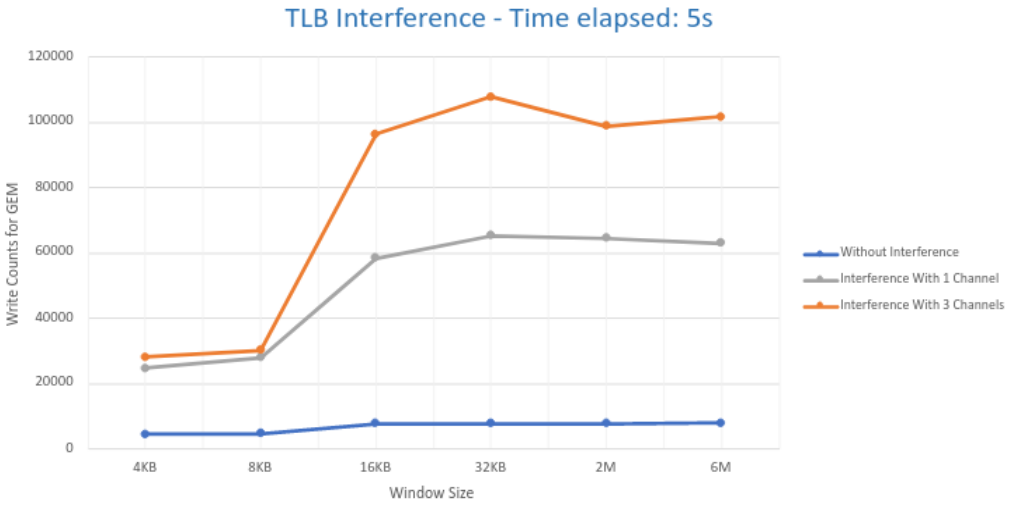


Figure 5.9: GEM device performance evaluation by counting the number of writes made by the GEM device in the TLB for different windows sizes.

Figure 5.9 presents the number of misses the GEM device has, when a bare-metal guest is generating interference from the LPD device's channel usage. This test followed the same settings as in subsection 5.4.1. Looking at Figure 5.9 and comparing it to Figure 5.4, it is possible to see that the number of misses skyrockets, just like in Figure 5.4. By directly comparing the two figures, it is observable that when using this new interference method that is more dedicated to interfering in the TLB than in the DRAM, the number of misses is aggressively much higher than the number of misses when the simple mode is used. Another conclusion drawn from these graphs is that it is noticeable that in Figure 5.9, there is a more significant difference between the two interference configurations. This is because using this new method where it simply has to transfer 16 bytes from each page in the array, instead of having to transfer the 4K bytes that the page has, makes the time for each transaction much shorter, and it is then possible to do even more access in turn.

Regulation 0x20

Now that it was seen that switching from simple mode to scatter-gather mode with a linked descriptor causes the number of misses to increase considerably, it remains to evaluate how this interference will behave when regulation is applied to the LPD channels. It will then start with a 0x20 regulation for each channel, as was done for the tests using simple mode.

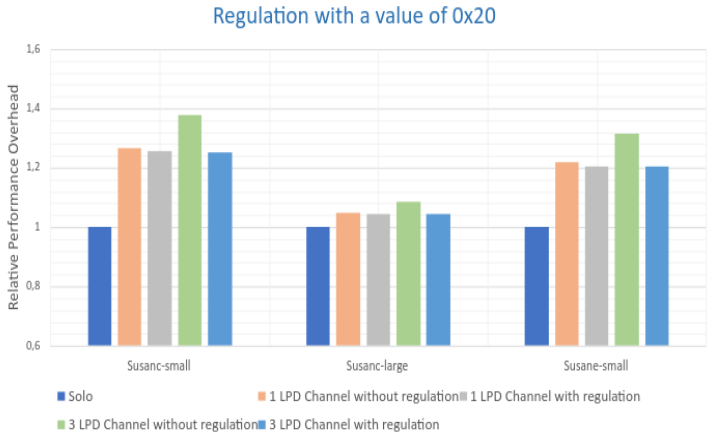


Figure 5.10: Performance Overheads of Mibench automotive benchmark when generating interference using LPD channels relative to a Linux guest execution with and without Regulation using the linked descriptor.

Figure 5.10 presents the results obtained from the automotive MiBench subset. Looking at Figure 5.10, something very interesting is exposed, which is if one takes this figure and compares it to the test where the same scenario as in this test is applied, but only the mode of operation of the LPD device channels is changed, in subsection 5.4.1. It is extracted that when regulation is applied in this scenario for the three different benchmarks, the interference does not drop considerably compared to the Figure 5.5. It is also notable that both configurations where regulation is applied reduce to the same performance overhead of one point twenty-five times compared to the solo Linux run.

It is now evaluated the system's behavior regarding the interference in the TLB, using the regulators to see the difference between the number of misses when the regulation is applied and not.

As in the previous tests that were performed to evaluate the number of misses of the TLB, the use of regulation reduces the impact that the GEM device suffers. It is true that just like the test that was performed for figure 5.10, for a configuration where it uses only one channel of the LPD device, a very significant reduction is not evident.

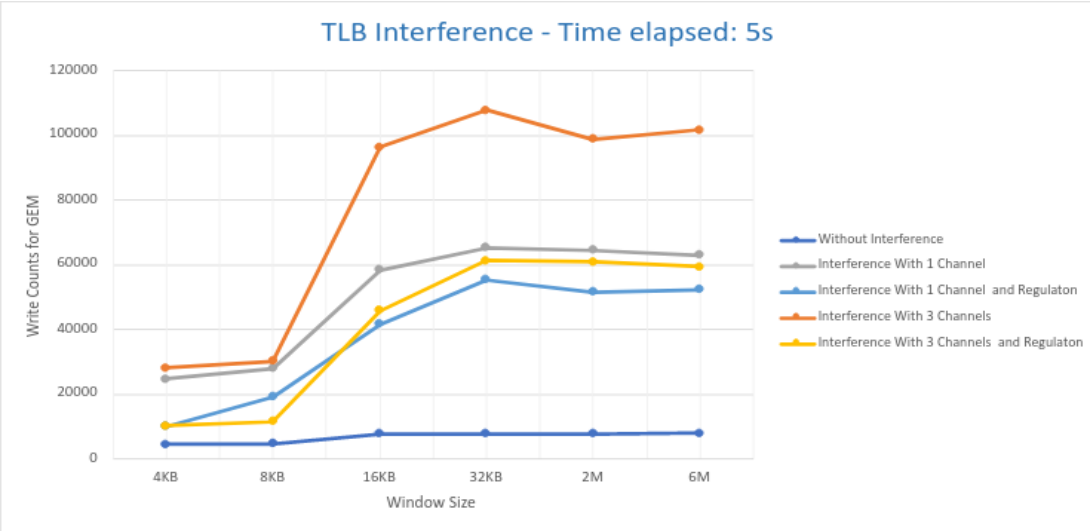


Figure 5.11: *Write* event count for GEM device,comparing the number of writes to the TLB when using regulation and not using the regulation.

The difference between the configurations where only one device channel is used to generate interference without and with regulation is not very big, although it is still evident. This lack of difference is due to changing from a type of interference that consistently manipulates the DRAM to a type of interference that is more directed to ARM the TLB, generating a lot more misses and no longer manipulating as much the DRAM. With this shift, the number of memory accesses is reduced, so the regulator does not act as much on the device.

Then the question arises as to how it is that in the test that was done with the help of the miBench automotive benchmark, it has so much performance overhead if the interference generated no longer makes so many accesses to memory. In turn, the guest should no longer have so many delays when the benchmark is run. This performance overhead comes from the delays that exist in the SMMU translations. If multiple misses are generated in the TLB, the SMMU must do a page table walk every time it wants to access memory. Looking at Figure 5.11, it is apparent that there is an actuation by the regulators. The two configurations that apply regulation stabilize relatively close to each other. However, even so, a regulation of 0x20 is still a long way from the configuration where there is no interference, especially compared to Figure 5.6.

Regulation 0x10

Now changing the regulation of the LPD device to 0x10. To have a term of comparison to the previous test and also to be able to compare with the tests realized in sub-subsection 5.4.1. Figure 5.12 presents the results obtained from the automotive MiBench subset.

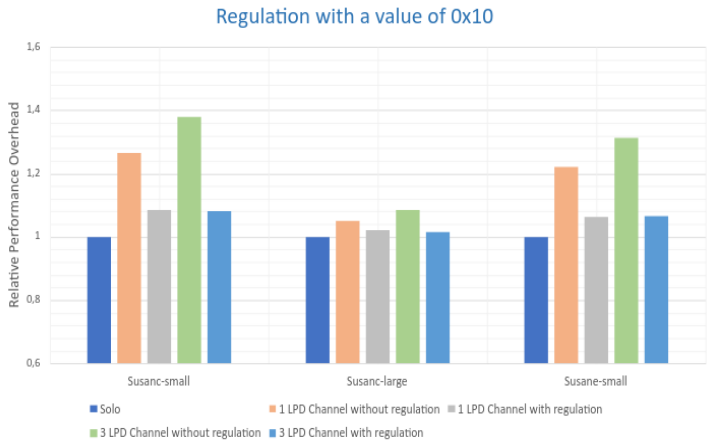


Figure 5.12: Performance Overheads of Mibench automotive benchmark when generating interference using LPD channels relative to a Linux guest execution with and without Regulation using the linked descriptor.

Analyzing Figure 5.12, it is evident that there was a significant difference to the test performed in Figure 5.10. As in the test of figure 5.10, the overhead performance of the Linux execution is stabilized at the same value for the two configurations that were added in order to evaluate the system response when the device is regulated.

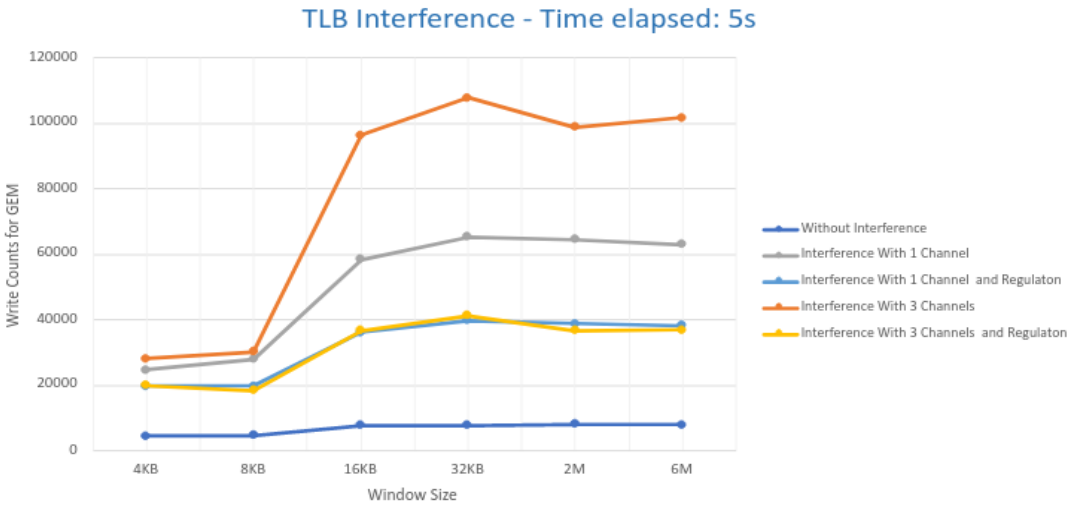


Figure 5.13: *Write* event count for GEM device,comparing the number of writes to the TLB when using regulation and not using the regulation.

Analyzing Figure 5.13, it can be concluded that the regulation of 0x10 significantly affects the number of misses that the GEM device has. For this regulation, there is a remarkable difference between the configuration where one interference channel is used to the two configurations where the regulation is applied. As concluded in subsection 5.4.1, it is also evident that the two configurations where regulation is used tend towards the same value. However, although a more significant effect is felt than for the regulation of 0x20, there is still a significant amount of misses that the GEM device has.

5.4.3 Bare-metal Benchmark

In the previous tests, it was observed that the use of DMA accelerators could generate a relative interference to another guest. However, it was also observed that the use of regulators for this same accelerator significantly reduces the interference that DMA can generate in both TLB and DRAM. What has not been evaluated is that when regulators are applied to a DMA device, the regulation is applied to every channel that this device has, which is a setback. This is because, as Bao was developed, it allows an initial config to be made in which it is specified what each VM has access to. Consider a scenario in which a guest is assigned a channel from the LPD device, and another channel is assigned to another guest. If some regulation is imposed, both guests will feel this regulation even if only one of these guests is creating interference in the TLB, which is a bit counterproductive.

To evaluate this scenario, a bare-metal guest was used as a benchmark to evaluate the time elapsed from the beginning to the end of a transfer. This test was evaluated for five different scenarios:

- The first scenario: Where the bare-metal benchmark is running without any interference to achieve a baseline;
- The second scenario: Where the bare-metal benchmark is running in parallel with a bare-metal guest with the intent of generating interference in the TLB;
- The third scenario: Where the second scenario is repeated but applies a regulation of 0x20;
- The fourth scenario: Where the second scenario is repeated but applies a regulation of 0x10;
- The fifth scenario: Where the second scenario is repeated but applies a regulation of 0x1.

For all scenarios, the same transfer of 16 bytes was performed, and to reduce the time variation of a transfer, it was taken the average time of one hundred transfers. Now retaining the average time of a 16 bytes transfer is possible to calculate the channel's bandwidth and gain a comparative measurement to the other scenarios. To better understand and observe the outcome of these tests it was extracted the number of writes and reads to the TLB. The percentage lost in Table 5.1 is relative to the LPD channel's bandwidth without interference and regulation.

Table 5.1: Bare-metal benchmark - tests outcome.

	Calculated Bandwidth (MB/s)	Bandwidth Percentage Lost (%)
Without Interference	10.238	0
With 3 LPD Channels Inter	7.183	29.28
With 3 LPD Channels Inter - Regulation 0x20	3.862	62.28
With 3 LPD Channels Inter - Regulation 0x10	2.374	76.81
With 3 LPD Channels Inter - Regulation 0x01	0.180	93.23

Analyzing Table 5.1 it is observable that, as expected, when interference is applied, the performance of the channel in question is affected since it and the interference channels share the same TLB.

Analyzing when the regulation is applied to the interference channels and to the channel in which the transfer time is evaluated, it is concluded that as the regulation value is lowered, that is, applying a greater regulation to the channels, the loss of the channel's performance in evaluation increases. This states that using regulators for scenarios that, as the Bao hypervisor idealizes, is not applicable. Obviously, this is a corner case because this would not be applicable in the scenario between the DMA and the GEM. After all, the DMA and the GEM do not share the same regulator. However, the scenario exploited in this section is something to be aware of.

6. Conclusion

Multiple fields have adopted multicore systems, and one of the main issues is the competition for hardware resources and the resulting interference brought on by the presence of many software systems on the same platform. The usage of a hypervisor, which enables the virtualization of resources to service several isolated domains running on the same platform, offers a promising solution to this problem. Numerous commercial and open-source hypervisors are available. However, most of them lack robust isolation features in the contention of cache levels and main memory bandwidth, which have been discovered to dramatically affect the timing aspects of the software executing on a multicore platform.

In the first stage of its development, this dissertation focused on designing and implementing an isolation mechanism for the DRAM controller of an ARM platform, with an emphasis on its integration with the Bao hypervisor. The virtualization technologies provided by Bao have been carefully integrated with the mechanism. The influence of the DMA accelerator on other domains and DMA accelerators is examined in a later stage of this dissertation.

ARM PMU have been utilized to integrate memory bandwidth isolation, allowing the hardware-based counting of each core's memory accesses. At this stage, two handlers have been set up using the Private Peripheral Interrupt (PPI) to handle CPU scheduling in the event of a budget overflow and to control a recurring budget recharge event that is initiated by a timer.

The efficacy and performance of the developed mechanism have been evaluated through experimental findings utilizing cutting-edge benchmarks. The results demonstrate a clear improvement in the hypervisor's isolation capabilities, increasing execution predictability without requiring prior knowledge of the software in the other domains.

Several technologies were used for the research of this second phase. Such as using DMA controllers for interference purposes in other guests or in other DMA accelerators. In order to verify interference at the level of the GEM module's DMA accelerator, the performance monitoring extension technology provided by SMMU is used. The final phase of this research is to use the QoS-400 regulators that the ZCU104 board provides and thus draw conclusions about whether these regulators are favorable for reducing interference at the guest and the DMAs level.

The interference generated by the DMAs is studied in detail through experiments using benchmarks and synthetic microbenchmarks to study the impact of the generated interference on the guest, as on other DMAs. The results show that interference generated by DMAs has a significant impact on other devices

as well as other guests. It is also concluded that despite the use of regulators, it has a significant impact on reducing interference when these are applied to other DMA accelerators. However, when different guests share the DMA LPD accelerator channels, this regulation affects both the channels that generate interference and the channels that do not.

6.1 Future Work

As the application of regulation does not fulfill the results expected by applying these regulators, other solutions were thought of. The TLB presents a source of unpredictability for real-time systems. Future work would be applying TLB coloring for the Bao hypervisor at the SMMU level. A problem that arises by implementing TLB coloring is the integration that would need to exist with Bao's cache coloring mechanism.

No solution prevents interference at the cache and TLB levels, despite the addition of coloring methods to the cache and TLB levels. For instance, two pages could map two different cache sets in the last cache level. However, it might map to the same TLB set, introducing interference and hence unpredictable behavior, so interference at both levels should be minimized to prevent a unified solution. Therefore, to prevent interference, two physical pages corresponding to two different cache colors should map to two different TLB colors. In conclusion, it requires the coexisting of the coloring mechanism for the cache and for the TLB.

References

[1] Bruno Sá, José Martins, and Sandro Pinto. A First Look at RISC-V Virtualization from an Embedded Systems Perspective. *IEEE Transactions on Computers*, PP:1–1, 11 2021.

[2] Paolo Modica, Alessandro Biondi, Giorgio Buttazzo, and Anup Patel. Supporting temporal and spatial isolation in a hypervisor for ARM multicore platforms. In *2018 IEEE International Conference on Industrial Technology (ICIT)*, pages 1651–1657, 2018.

[3] Samuel Pereira, João Sousa, Sandro Pinto, José Martins, and David Cerdeira. Bao-Enclave: Virtualization-based Enclaves for Arm. *The IEEE World Forum on the Internet of Things*, 09 2022.

[4] Asif Iqbal, Nayeema Sadeque, and Rafika Mutia. An Overview of Microkernel, Hypervisor and Microvisor Virtualization Approaches for Embedded Systems. *Report, Department of Electrical and Information Technology, Lund University, Sweden*, 10 2022.

[5] Alexandra Aguiar and Fabiano Hessel. Embedded systems' virtualization: The next challenge? In *Proceedings of 2010 21st IEEE International Symposium on Rapid System Protyping*, pages 1–7, 2010.

[6] Steven Hand, Andrew Warfield, Keir Fraser, Evangelos Kotsovinos, and Daniel Magenheimer. Are Virtual Machine Monitors Microkernels Done Right? *Operating Systems Review - SIGOPS*, 40, 01 2005.

[7] Jordan Shropshire. Analysis of Monolithic and Microkernel Architectures: Towards Secure Hypervisor Design. In *2014 47th Hawaii International Conference on System Sciences*, pages 5008–5017, 2014.

[8] Pagare Jayshri and Nitin Koli. A technical review on comparison of XEN and KVM hypervisors an analysis of virtualization technologies. *IJARCCE*, pages 8828–8832, 12 2014.

[9] Anup Patel, Mai Daftedar, Mohamed Shalan, and M. Watheq El-Kharashi. Embedded Hypervisor Xvisor: A Comparative Analysis. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 682–691, 2015.

[10] José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. *In Workshop on Next Generation Real-Time Embedded Systems*, 01 2020.

[11] ARM. *ARM® Cortex™-A Series*. 2011.

[12] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. University of Wisconsin, 2018.

[13] ARM. *ARM Cortex-A53 MPCore Processor Technical Reference Manual*. 2011.

[14] Qing Li, David Hartley, and Brian Rosenberg. An Introduction to Access Control on Qualcomm Snapdragon Platforms. *Qualcomm Technologies, Inc*, 09 2020.

[15] Kyriakos Paraskevas, Konstantinos Iordanou, Mikel Lujan, and John Goodacre. Analysis of the Usage Models of System Memory Management Unit in Accelerator-attached. 2020. 6th International Symposium on Memory Systems ; Conference date: 28-09-2020 Through 01-10-2020.

[16] Stage 2 translation. https://developer.arm.com/documentation/102142/0100/Stage-2-translation. Accessed: 2021–11-28.

[17] ARM. *Arm® CoreLink™ MMU-600 System Memory Management Unit Technical Reference Manual*. 2016.

[18] José Carvalho Martins. Ontology-Driven Metamodeling Towards Hypervisor Design Automation: Microkernel Infrastructure. Master's thesis, Universidade do Minho, 2018.

[19] José Martins, João Alves, Jorge Cabral, Adriano Tavares, and Sandro Pinto. RTZVisor: A Secure and Safe Real-Time Hypervisor. *Electronics*, 6:93, 10 2017.

[20] Robert Kaiser. Complex embedded systems - A case for virtualization. In *2009 Seventh Workshop on Intelligent solutions in Embedded Systems*, pages 135–140, 2009.

[21] C. Scordino, L. Cuomo, M. Solieri, and M. Sojka. HERCULES: High-Performance Real-Time Architectures for Low-Power Embedded Systems; Multi-OS Integration and Virtualization. 2018.

[22] Embedded Systems. https://www.omnisci.com/technical-glossary/embedded-systems. Accessed: 2021–11-12.

[23] Difference between Full Virtualization and Paravirtualization. https://www.geeksforgeeks.org/difference-between-full-virtualization-and-paravirtualization/. Accessed: 2021–11-12.

[24] Paolo Modica. Temporal and Spatial Isolation in Hypervisors for Multicore Real-Time Systems. Master's thesis, Università Di Pisa, 2017.

[25] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37:164–177, 01 2003.

[26] José Martins and Sandro Pinto. Bao: a modern lightweight embedded hypervisor. *In Embedded World 2020 Exibition and Conference*, 02 2020.

[27] Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. Look Mum, no VM Exits! (Almost). *In: Proc. of the 13th OSPERT*, 05 2017.

[28] James O'toole, Dawson Engler, and M. Kaashoek. Exokernel: an operating system architecture for application-level resource management. 01 1995. Conference: ACM Symposium on Operating Systems Principles (SOSP).

[29] Quentin Perret, Pascal Maurere, Eric Noulard, Claire Pagetti, Pascal Sainrat, and Benoit Triquet. Temporal Isolation of Hard Real-Time Applications on Many-Core Processors. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11, 2016.

[30] Steven H. VanderLeest and David C. Matthews. Incremental Assurance of Multicore Integrated Modular Avionics (IMA). *2021 IEEE/AIAA 40th Digital Avionics Systems Conference (DASC)*, 2021.

[31] J. Liedtke, H. Hartig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Proceedings Third IEEE Real-Time Technology and Applications Symposium*, pages 213–224, 1997.

[32] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. *Real-Time Technology and Applications - Proceedings*, 2014:155–166, 10 2014.

[33] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. 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)*, pages 55–64, 2013.

[34] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory Bandwidth Management for Efficient Performance Isolation in Multi-core Platforms. *IEEE Transactions on Computers*, 2015.

[35] Matteo Zini, Giorgiomaria Cicero, Daniel Casini, and Alessandro Biondi. Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms. *Software: Practice and Experience*, 52, 11 2021.

[36] Matthew Guthaus, Jeffrey Ringenberg, Daniel Ernst, Todd Austin, Trevor Mudge, and Richard Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Conference: Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3 – 14, 01 2002.

[37] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2016.

[38] Shrinivas Anand Panchamukhi and Frank Mueller. Providing task isolation via TLB coloring. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 3–13, 2015.