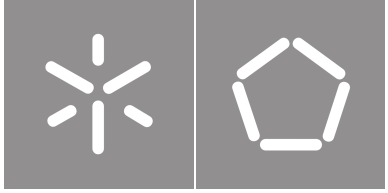
Universidade do Minho
Escola de Engenharia

Gonçalo Gonçalves Freitas

Exploring the Arm MPAM Extension for Static Partitioning Virtualization

Março de 2023



Universidade do Minho
Escola de Engenharia

Gonçalo Gonçalves Freitas

Exploring the Arm MPAM Extension for Static Partitioning Virtualization

Dissertação de Mestrado
Mestrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação de:
Professor Doutor Sandro Pinto

Março de 2023

Agradecimentos

Esta dissertação é a representação do fim de toda uma jornada académica e do início de um novo capítulo. Como tal, gostaria de agradecer às pessoas que sempre me apoiaram e prestaram um algum do seu tempo e experiência, tornando isto possível.

Ao meu orientador, Professor Doutor Sandro Pinto, um sincero obrigado por todo o conhecimento transmitido, mas acima de tudo pela confiança depositada para a realização deste trabalho. Agradeço também aos meus amigos de curso, que foram um grande apoio em todo o meu percurso académico. Um agradecimento especial para as pessoas com quem tive o prazer de conviver e partilhar experiências diariamente - Diogo Matias, Francisco Dias, Vitor Ribeiro e Samuel Pereira. Também, aos colegas e amigos - Mestre Diogo Costa e Mestre João Sousa - por toda a ajuda dada nos momentos mais dificeis e a todos que fizeram parte deste meu percurso de desenvolvimento, um sincero obrigado.

Por fim, quero agradecer profundamente aos meus pais, que fizeram com que nesta jornada nunca me faltasse apoio, motivação e amor que deram ao longo destes anos.

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

Exploração da Extensão Arm MPAM para a Virtualização em Partições Estáticas

Os sistemas embebidos desempenham um papel crucial na garantia da segurança dos sistemas de transporte e controlo industrial. Estes sistemas, conhecidos como sistemas críticos de segurança, podem ter consequências catastróficas em caso de falha. Nos últimos anos, e principalmente pressionados por restrições SWaP-C, os sistemas embebidos consolidaram funções com diferentes níveis de criticidade, criando uma nova classe de sistemas chamados sistemas de criticidade-mista.

A virtualização ganhou popularidade nos sistemas de criticidade-mista, pois permite a consolidação e isolamento de múltiplos domínios numa mesma plataforma. Os hipervisores proporcionam isolamento temporal e espacial, e as garantias *freedom-from-interference* satisfazem as normas de segurança. No entanto, as plataformas modernas *multicore* também trazem dificuldades devido à interferência causada por recursos de hardware partilhados (por exemplo, LLC, *bus*, e memória principal). Sem uma gestão adequada, estas interferências podem prejudicar a previsibilidade temporal. Para responder a estes problemas, foram propostos mecanismos tais como a partição de cache e a reserva de largura de banda de memória. Além disso, o recente lançamento da especificação da Arm, MPAM, que fornece novas oportunidades para a regulação do acesso à memória.

Esta dissertação apresenta o design e implementação de um mecanismo de reserva de largura de banda de memória (designado safe_mem), desenvolvido no hipervisor Bao. O safe_mem fornece reserva de largura de banda limitando o acesso à memória por CPU a cada período, assegurando o isolamento temporal no sistema. Esta dissertação também descreve o design e implementação de uma API para a extensão MPAM, que inclui mecanismos de controlo de partição de cache. Por último, apresenta uma análise relacionada com a contenção da memória e verifica a eficácia dos mecanismos implementados para os processadores Arm Cortex-A.

Keywords: Memory hierarchy, Safety, Shared cache, Multicore, Real-time systems, Bandwidth reservation, Cache partitioning, Memory-access predictability, Isolation, Arm Cortex-A.

Abstract

Exploring the Arm MPAM Extension for Static Partitioning Virtualization

Embedded systems play a crucial role in ensuring the safety of transportation and industrial control systems. These systems, known as safety-critical systems, can have catastrophic consequences in case of failure. In recent years, and mainly pushed by Size, Weight, Power and Cost (SWaP-C) constraints, embedded systems have consolidated functions with different criticality levels to create a new class of systems called Mixed Criticality System (MCS).

Virtualization has gained popularity in MCS as it enables the consolidation and isolation of multiple environments onto the same platform. Hypervisors provide temporal and spatial isolation, i.e., the freedom-from-interference guarantees required from safety standards. However, modern multicore platforms bring difficulties due to the interference caused by shared hardware resources (e.g., Last-Level Cache (LLC), system bus, and main memory). Without proper management, these interferences can harm timing predictability. To address these issues, mechanisms such as cache partitioning and memory bandwidth reservation have been proposed. Also, Arm's recent release of the Memory System Resource Partitioning and Monitoring (MPAM) specification provides new opportunities for memory access regulation.

This dissertation presents the design and implementation of a memory bandwidth reservation mechanism (named safe_mem) developed on top of the Bao hypervisor. The safe_mem provides bandwidth reservation by limiting the memory access for each Central Process Unit (CPU) every period, ensuring temporal isolation in the system. The dissertation also describes the design and implementation of an Application Programming Interface (API) for the MPAM extension, which includes cache partitioning control mechanisms. Lastly, presents an analysis related to memory contention and verifies the effectiveness of the implemented mechanisms for Arm Cortex-A processors.

Keywords: Memory hierarchy, Safety, Shared cache, Multicore, Real-time systems, Bandwidth reservation, Cache partitioning, Memory-access predictability, Isolation, Arm Cortex-A.

Contents

List of Figures

List of Tables

Glossary

AEM Architecture Envelope Models

API Application Programming Interface

APU Accelerated Processing Unit

BSP Board Support Package

COTS Commercial-Off-The-Shelf

CPU Central Process Unit

DoS Denial of Service

DRAM Dynamic Random-Access Memory

DTS Device-Tree Source

ECU Eletronic Control Unit

FR-FCFS First-Ready First-Come First-Serve

FVP Fixed Virtual Platform

GIC Generic Interrupt Controller

GPOS General Purpose Operating System

HAL Hardware Abstraction Layer

IPC Inter-Processors Communications

ISA Instruction Set Architecture

IT Information Technology

LLC Last-Level Cache

MCS Mixed Criticality System

MCU Micro-Controller Unit

MMR Memory-Mapped Register

MMU Memory Management Unit

MPAM Memory System Resource Partitioning and Monitoring

MSC Memory System Component

OS Operating System

PE Processing Element

PMU Performance Monitor Unit

QoS Quality of Service

RTOS Real Time Operating System

SCU System Control Unit

SMMU System Memory Management Unit

SoC System on Chip

SWaP-C Size, Weight, Power and Cost

TCB Trusted Computing Base

VM Virtual Machine

VMM Virtual Machine Monitor

WCET Worst-Case Execution Time

1. Introduction

The trend of increasing computational power in domains like automotive and industrial control [1], made possible by modern computer architectures, has conducted to the integration of a large number of complex and resource-intensive applications onto a single hardware platform. This demand for high-performance embedded systems has been rapidly growing in recent years and is expected to continue increasing in the near future [2].

Despite the SWaP-C limitations, it is no longer possible to implement each function in a dedicated platform. Market pressure has resulted in the integration of multiple subsystems with different levels of criticality onto a single platform, known as MCS. These systems must maintain real-time requirements while balancing the conflicting needs of isolation for security and safety, and efficient resource sharing [3]. This market pressure has led to a shift from the use of small, single-core Micro-Controller Units (MCUs) running bare-metal applications or Real Time Operating Systems (RTOSs) to powerful multicore platforms with complex memory hierarchies that can host General Purpose Operating Systems (GPOSs) [4]. In this regard, it is fundamental to provide isolation between applications with different criticality levels in order to avoid contention for shared computational resources [5], such as main memory and shared caches, which can result in significant performance degradation and lack of determinism [6].

Virtualization, a well-established technology in the cloud space, has emerged as a natural solution for consolidating distinct subsystems with different criticality levels onto the same hardware platform [7, 8]. It requires minimal engineering efforts to support legacy software while ensuring separation and fault containment between different Virtual Machines (VMs). Virtualization technology leverages a Virtual Machine Monitor (VMM), also known as a hypervisor, to provide isolation between multiple subsystems. One key aspect of this isolation is temporal isolation, which controls the allocation of resources, such as CPU time, among the various subsystems. This is achieved through the implementation of scheduling algorithms that distribute time slots for resource access among the subsystems. Another important aspect of isolation provided by virtualization is spatial isolation. This refers to the practice of dividing resources into distinct and separate partitions, which are then allocated to specific subsystems. This approach ensures that each subsystem can only access the resources allocated to its partition, preventing any conflicts or interference with other subsystems. The implementation of spatial isolation serves to enhance the security, stability, and performance of the system, enabling multiple VMs to operate as if they were running on separate physical hardware [8].

While virtualization technology offers guarantees of spatial isolation, there are still challenges in achieving temporal isolation when consolidating MCSs. Many existing VMMs have been found to have insufficient capabilities in addressing architectural constraints and real-time requirements. This is despite the fact that they typically virtualize primary hardware resources such as CPUs, memory, and I/O devices. However, certain hardware resources (e.g., LLC, the system bus, and the main memory) are shared among the different systems. The mitigation of interference on multicore platforms is typically done by leveraging two techniques: (i) cache partitioning and (ii) memory bandwidth reservation.

Cache partitioning. Cache partitioning is a technique used to reduce cache misses in computer systems. A cache miss occurs when two or more different memory addresses map to the same cache line, causing one of the addresses to be evicted from the cache, even though it is still needed. Typically, cache partitioning is implemented either via cache coloring or cache locking. Cache coloring is a technique that assigns different colors to different memory blocks, and assigns cache lines to specific colors. This way, the same color can not be assigned to different memory blocks, reducing the number of cache misses. Cache locking, on the other hand, is a technique that prevents specific cache lines from being evicted from the cache, allowing the data to remain in the cache for an extended period of time.

Memory bandwidth reservation. Memory bandwidth reservation is a technique that allows a system to reserve a certain amount of memory bandwidth for a specific task or application. When applications execute concurrently on a multicore platform and compete for access to the same limited-bandwidth shared resource, it can significantly hamper the system performance [9]. Therefore, managing memory bandwidth is essential to prevent interference between different domains that are competing for access to the memory bus, which greatly degrades both system and individual application performance [10, 11].

Besides the traditional approach to implementing cache and memory bandwidth partitioning, Arm recently released MPAM processor extension specification for Armv8-A. The MPAM specification outlines that various memory subsystem components, such as caches, interconnects, and Dynamic Random-Access Memory (DRAM) memory controllers may support this processor extension. However, the specification is quite detailed, so this dissertation will explore the features and capabilities of the MPAM extension to propose a potential implementation for managing memory resources, specifically for cache and memory bandwidth partitioning.

1.1 Aim and Scope

The primary goals of this dissertation are the design and development of a software mechanism that mitigates interference between shared resources on the same platform by controlling the memory bandwidth used by each VM. Additionally, this dissertation involves the configuration of the hardware processor extension, MPAM, through an API. This extension prevents a single process running on a CPU from monopolizing a cache and other shared resources, providing temporal and spatial isolation between the domains of the system [6].

The main goal of these hardware-supported virtualization technologies is to mitigate interference between VMs in shared resources such as the LLC, system bus, and DRAM controller on the same platform. Additionally, the proposed software mechanism, safe_mem, is designed to control access to main memory from CPUs by configuring and monitoring the maximum number of accesses for each CPU in each period through the Performance Monitor Unit (PMU). If the selected PMU event exceeds the predefined value, access to memory is limited to mitigate interference in the LLC shared by all system domains. Also, MPAM API will also be able to control cache usage and bandwidth allocation for each VM, providing inter-VM temporal and spatial isolation. This evaluation will focus on finding a configuration that optimizes the performance of Arm Cortex-A architecture. However, the Bao hypervisor's software mechanisms must have a modular architecture to ensure compatibility with other computing architectures and scalability with the increasing number of connected devices.

These mechanisms follow the above cited set of goals, and the main objectives for this work are outlined below:

- Design and development of a software mechanism to control and monitor bandwidth usage by each CPU through statically defined configuration parameters;
- Design and development of an API for the Arm processor extension, MPAM, to control and monitor directly the hardware limiting its usage by each VM.

1.2 Dissertation Structure

This summary covers the main chapters of this document. Chapter 1 introduces the research question and objectives of this dissertation. Chapter 2 is divided into four sections. The first section provides an overview of embedded systems, including soft and hard real-time systems and the spatial and temporal isolation challenges in MCSs. The second section covers the key concepts of virtualization and hypervisors, including architecture types, design, and types of virtualization. The third section discusses interference in multicore platforms, its sources, and techniques for mitigation through the cache and DRAM partitioning. Finally, the final section presents relevant academic research related work.

Chapter 3 describes the platforms and tools, and is divided into two sections because two different platforms, a hardware, and a software model, were used to run each mechanism due to the unavailability of the MPAM extension at the time of writing. The first section discusses the specifications of some Armv8-A boards. The second presents the Arm Fast Models specifications, including the Fixed Virtual Platform (FVP) model used to implement and configure the MPAM extension. It also lists the benchmarks used to test and verify the implementation of both mechanisms.

Chapter 4 covers the design and development of both implementations of the proposed mechanism, safe_mem, and the MPAM API implemented in the Bao hypervisor to mitigate interference between domains with different criticality levels.

Chapter 5 evaluates the results of the mechanisms implemented on the hypervisor and identifies the best-case scenario for improving system performance. Finally, Chapter 6 discusses the results of each implemented method for mitigating interference, highlighting the limitations of each approach and suggesting potential future developments to address these limitations in the ”Future Work”, section 6.2.

2. Background and State of the Art

Embedded systems, which were once simple and single-purpose due to real-time demands and hardware resource limitations, are becoming more complex and multifunctional, resulting in a higher number of bugs and vulnerabilities [12, 13]. To ensure safety and security in the face of massive failures or malicious attacks [14, 15], virtualization has emerged as a natural solution through the isolation and fault containment provided by encapsulating each embedded subsystem in its own VM. This technology also allows the consolidation of different applications onto a single hardware platform, reducing size, weight, power, and cost (SWaP-C) [16]. Modern Commercial-Off-The-Shelf (COTS) heterogeneous architectures with multi-core accelerators can provide energy-efficient performance through the integration of multiple computing elements running at lower frequencies, while architectural heterogeneity enhances platform flexibility [1].

This section provides an overview of the state-of-the-art, fundamental concepts, and relevant work related to this dissertation. It covers background knowledge on embedded systems, section 2.1, virtualization and hypervisor concepts, section 2.2, interference in multicore platforms, section 2.3, and recent work on memory management to mitigate interference, section 2.4.

2.1 Embedded Systems

2.1.1 Soft Real-Time Systems

Real-time systems with soft timing constraints are becoming more common as embedded systems become widespread. In these systems, tasks can still function effectively even if deadlines are not guaranteed to be met. As a result, temporary deadline misses may degrade the Quality of Service (QoS), but they will not cause inappropriate system behavior. Furthermore, the systems referred to as soft real-time include a combination of varying properties, all of which share the common property that resource allocation and dispatching requirements are looser relative to hard real-time [17]. Recently, various research efforts have focused on the design and optimization of real-time systems [18]. These systems require unique design considerations due to timing constraints placed on the tasks. Even though these real-time tasks are time-bound, their timing constraints are not based on absolute values. In summary, these types of systems are used in non-critical safety applications such as multimedia transmission and reception, as

their tasks do not strictly require to complete executed within a specific deadline. This can result in system performance degradation, but the application will continue to run. Overall, this type of real-time system has more flexibility because it can tolerate some level of deadline misses.

2.1.2 Hard Real-Time Systems

Hard real-time systems are a class of real-time systems where the consequences of missing a deadline are severe, such as injury or death. These systems are characterized by strict timing constraints and high-reliability requirements. In hard real-time systems, the timing constraints are defined in terms of the Worst-Case Execution Time (WCET) of the system's tasks, which is the maximum time that a task can take to execute under any possible input and any possible initial state of the system. The WCET is a crucial parameter for hard real-time systems, as it determines the maximum amount of time that the system can tolerate for a task to complete [19].

In order to meet the timing constraints, hard real-time systems typically use a combination of techniques, such as time-triggered scheduling, static priority scheduling, and rate-monotonic scheduling. These scheduling algorithms ensure that the system's tasks are executed in a predictable and deterministic manner, even in the presence of system perturbations, such as interrupts or external events. In addition to scheduling, hard real-time systems also use other techniques to improve the system's timing predictability (e.g., implement cache and memory bandwidth partitioning, use RTOS or even use specialized hardware). Due to the high reliability and strict timing requirements, hard real-time systems are typically more difficult to design and implement than soft real-time systems. However, they play a crucial role in ensuring the safety and reliability of critical systems such as the aerospace, automotive, and industrial control industries.

2.1.3 Mixed Criticality Systems

In recent years, the number of functional requirements in automotive and industrial control domains has increased significantly [20]. As a result, there is a growing demand for high-performance embedded systems to handle the increasing complexity of these systems and power-hungry applications [4]. These systems can run a range of applications with varying criticality, such as safety-critical and non-safety-critical. When consolidating these systems, it is necessary to balance the conflicting requirements of isolation for security, safety, and efficient resource sharing between domains, due to interference caused by contention for shared computational resources (e.g., processor, memory, bus, I/O devices). These types of systems are typically embedded in machines whose safety is critical, such as network-connected infotainment in automotive safety-critical control systems [21].

In conclusion, this type of system runs a variety of applications with different levels of safety criticality, which can generate interference when consolidated on shared resources. However, this interference can be mitigated, by improving system performance, through the usage of cache and DRAM partitioning

techniques. These techniques, as well as the sources of interference, are explained in more detail in section 2.3, which outlines the methods used to control this isolation and manage system resources while maintaining performance.

2.2 Virtualization and Hypervisors

Virtualization has emerged in embedded systems as complexity and real-time demands have increased, moving from single-functionality and single-purpose systems to more sophisticated and complex systems [22, 23]. It allows for the execution of multiple Operating Systems (OSs) on the same platform through techniques such as hardware and software partitioning or aggregation, partial or full machine simulation, emulation, time-sharing, and others. In short, because embedded systems have resource limits, real-time demands, and high-performance, the virtualization technique match virtual resources with physical resources, allowing hardware components to be abstracted and utilized as virtual resources by the software [24, 25], and uses native hardware for operations in the VMs [26].

To integrate and consolidate MCSs on a single hardware platform, virtualization technology is a viable solution, utilizing hypervisors such as Xen [27], KVM [28], and Bao [4]. Hypervisors, also known as VMM, act as an additional layer of software that manages the hardware resources used by VMs and creates an execution environment that is as similar as possible to the environment available when running on independent hardware, without coexisting systems [15, 29]. The software executed on a VM is separated from the hardware resources by the hypervisor or VMM, which is responsible for creating and managing the VMs running in the system, ensuring isolation between the multiple domains [30].

2.2.1 Types of Virtualization

Conceptually, a VM is an operating environment for a set of user-level applications, including libraries, the system call interface/service, system configurations, and in case it is not a baremetal also sets daemon processes, and file system state. The choice of the abstraction level at which virtualization is implemented involves trade-offs in terms of implementation complexity, run-time performance overhead, flexibility, and degree of isolation. In real-time systems, there are two major types of virtualization (Table 2.1): (i) para-virtualization and (ii) full-virtualization.

Para-virtualization Requires modifications to the kernel of the guest operating system, which communicates with the VMM through hypercalls rather than relying on the complete emulation of the system's hardware. While this approach poses some security risks, it offers better performance than full virtualization. The necessary adaptations to the OS are limited to the Hardware Abstraction Layer (HAL) or Board Support Package (BSP), which are required to port the OS to a different hardware platform [24, 25].

Full virtualization. Enables the hosting of an unmodified OS by relying on the hypervisor to emulate the low-level features of the underlying hardware as expected by the kernel of the guest OS, providing

a complete VM abstraction [25]. The hypervisor identifies privileged, control, and behavior-sensitive instructions that are trapped and emulated using the binary translation method, allowing direct execution of non-sensitive instructions. This decouples the guest OS from the underlying hardware but can be less optimal in terms of performance due to the time-consuming nature of the binary translation process. Multiple hypervisors such as KVM [28], Xen [27], and HYPER-V [31] explore this virtualization technique.

	Advantages	**Disadvantages**
Para-virtualization	No need total hardware emulation. Virtualized OS can directly communicate with hardware resources.	Guest OS need modifications Isolation is lower
Full-virtualization	Controls VMs access to system resources. No need of guest OS modifications.	Decrease in performance (everything emulated)

Table 2.1: Types of Virtualization: Advantages/Disadvantages.

In conclusion, while para-virtualization has the advantage of improved performance, reduced complexity in the hypervisor, and virtualization requirements on the architecture, it also has the limitation of incurring a high engineering cost associated with adapting each supported operating system to the hypervisor's unique platform interface. On the other hand, full virtualization requires the emulation of everything and synchronization between hardware and software resources, which is controlled by the need to virtualize, degrading system performance.

2.2.2 Hypervisor Architecture

Virtualization separates applications from the specific hardware characteristics they use to perform their tasks, creating a virtual environment to improve resource utilization [32]. The virtual environment is created by the hypervisor layer, which sits between the OS and the underlying hardware, divided into two categories: (i) type 1, or baremetal hypervisors, and (ii) type 2, or hosted hypervisors.

Type 1 hypervisor is a layer of software that is implemented directly on top of the underlying hardware, as shown in Figure 2.1. It provides device drivers that the guest OSs use to access the hardware. Because there is no software or other OS between the hypervisor and the hardware, this type of hypervisor provides excellent performance and stability. Its responsibility is to schedule and allocate system resources to VMs. Examples of native hypervisors include Xen [32], and Bao [4].

Type 2 hypervisor runs as an application within a normal OS, known as the host OS. The host OS treats the hypervisor as a system process. However, the added layer of the host OS can cause delays when a guest needs to use its resources, reducing system performance and increasing the risk of system failure due to host OS faults or malfunctions. Despite these performance and security concerns, hosted hypervisors are more affordable and suitable for software testing, and can be found in environments such as Oracle VM VirtualBox [25].

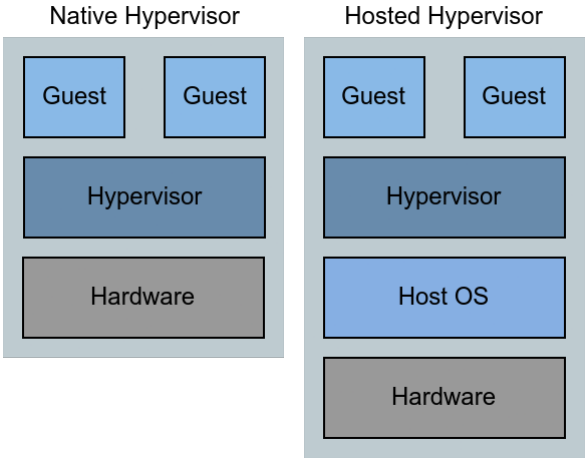


Figure 2.1: Hypervisor Architectures.

Hypervisor Design

Hypervisors are classified based on three factors: (i) the type of virtualization (as described in Section 2.2.1), (ii) the position of the hypervisor in the system stack, and (iii) the internal design of the hypervisor. There are two categories of hypervisor design, based on how they are implemented: (i) monolithic and (ii) microkernelized. This classification is important in terms of the performance measures achieved by the guests and the effectiveness of virtualization management.

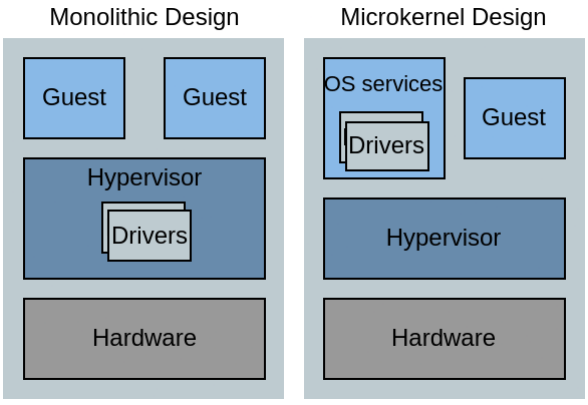


Figure 2.2: Hypervisor Designs.

Monolithic hypervisors control the hardware components for their partitions (e.g., inter-process communication, memory management, and I/O device drivers) by accessing the host hardware, virtualizing the CPU, and emulating guest I/O. Monolithic hypervisors have a large attack surface due to their support for many drivers. This means that a fault or error in the hypervisor code or third-party drivers (that it loads) could compromise the entire system. However, monolithic systems perform better in terms of inter-VM communication because all components are stored in the same address space, resulting in fewer context switches [33].

Microkernelized hypervisors are usually small microkernels that manage the minimum necessary components, such as inter-process communication and scheduling, and provide CPU virtualization and basic host hardware access. The OS functions such as the virtual memory manager, file system, and CPU scheduler, are built on top of the hypervisor. Each service and application has its own address space, providing protection between applications, OS services, and the kernel. As a result, when a component fails, only the host-guest is compromised while the other guests continue unaffected. However, due to the several introduced context switches in inter-VM communication, this design performs worse when compared with the monolithic approach [33].

The various types of hypervisors, designs, and virtualization techniques each have their own benefits and drawbacks, depending on the desired control over the VMs and the final system requirements. For instance, a hypervisor with a focus on safety and security should run on top of hardware to have full control over resources and use a microkernel design to minimize the attack surface and reduce the risk of malfunction. In terms of virtualization technique, VMs should be used in a balanced way that allows each guest to run independently and isolated from others, while not requiring a significantly higher Trusted Computing Base (TCB).

2.2.3 Static Partitioning Hypervisor

This architecture leverages hardware-assisted virtualization to implement a minimal software layer that statically partitions all platform resources and assigns each one exclusively to a single VM. It assumes no hardware resources need to be shared among guests. Since each virtual core is statically pinned to a single physical CPU, there is no need for a scheduler, and the hardware virtualization extensions allow for a minimal software layer, minimal TCB and virtualization overhead (VM interrupt latency, VM boot time), thus reducing size and complexity. While static partitioning may not be as efficient in micro-architectural resources usage, it provides stronger guarantees of isolation and real-time performance.

Despite the CPU and memory isolation provided by the static partitioning approach, it is not sufficient as many micro-architectural resources such as LLC, interconnects, and memory controllers remain shared among different domains, leading to a lack of temporal isolation, resulting in poor performance and determinism [34]. Also, this can be exploited by a malicious VMs to launch Denial of Service (DoS) attacks by increasing their consumption of a shared resource [20]. To address this issue, techniques such as cache partitioning (via locking or coloring) or memory bandwidth reservations, 2.3, have been proposed and implemented at Bao hypervisor [2].

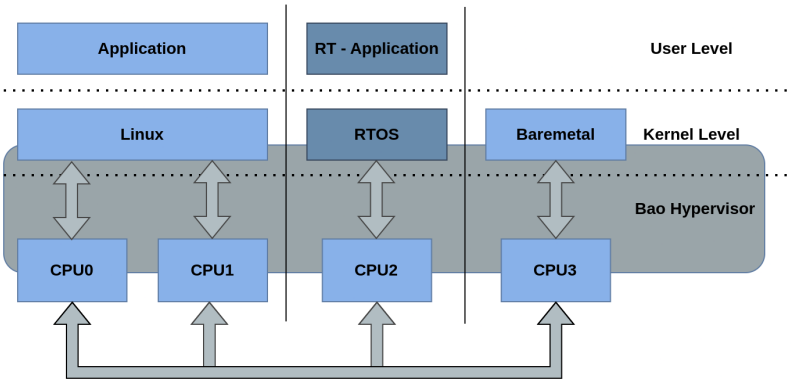


Figure 2.3: Bao Architecture.

2.3 Interference in Multicore Platforms

System performance can be affected by unpredictable memory requests processing time due to contention/interference in the shared memory hierarchy [30]. This section covers the basic concepts of interference on multicore systems, including its sources and techniques that mitigate them.

2.3.1 Sources of Interference

Many workloads in multicore systems have strict real-time requirements, and interference caused by contention for resources such as the LLC and memory controller, shared among partitions, generates non-determinism and unpredictability in the execution time of tasks, making it difficult to meet real-time requirements.

The CPU cache memory hierarchy is one of the main factors causing unpredictability. The magnitude of temporal unpredictability that arises from spatial contention in the cache hierarchy is typically larger than can be observed due to temporal contention. This way, the execution time of each task in a multicore system depends on the behavior of the cache hierarchy because of the existing conflicts in the allocation of cache lines (spatial contention) leading to different types of interference, including intra-task, intracore, and intercore interference. Intra-task interference occurs when two tasks have working sets larger than a specific cache level. Intracore interference occurs when a task evicts data from another task already stored in cache memory, increasing its access time. Intercore interference occurs in the LLC, which is shared among all system partitions, when it is accessed at the same time by multiple CPUs, leading to unpredictability due to repeated eviction of each other's data from the cache [35].

Similarly, issues can occur while accessing the DRAM memory because the memory controller, which manages the flow of data going to and from the system's main memory, is also a single and shared component.

2.3.2 Interference Mitigation Techniques

Interference mitigation techniques are a crucial aspect of real-time systems that use multicore platforms. These techniques are implemented to alleviate the negative effects of interference on system performance, in order to meet the strict real-time requirements of the workloads running on these systems. To address this, interference mitigation techniques aim to minimize contention, improve resource allocation efficiency and enhance the predictability of the system. One of the most widely used interference mitigation techniques is cache partitioning, which divides the LLC into smaller partitions, reducing contention among domains. Another technique is memory bandwidth reservation, which aims to reserve a specific amount of memory bandwidth for a domain or group of domains, thus improving memory access patterns and reducing contention for memory resources.

Cache Partitioning

Cache partitioning is typically implemented either via cache coloring or locking, which are well-established techniques to minimize interference generated by shared caches, LLC.

Cache coloring is a technique that assigns specific system partitions to cores by dividing the cache into partitions. This method aims to reduce inter-core interference, increase predictability, and ease WCET estimation [35].

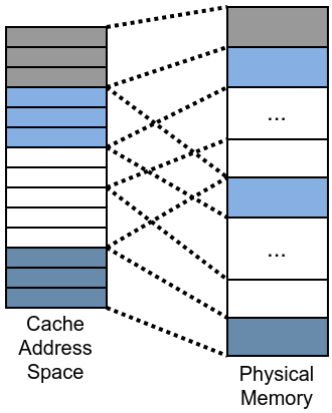


Figure 2.4: Mapping of Cache Coloring Technique.

The technique works by coloring physical frames such that two different colors will not map to the same cache set, reducing interference in physically indexed caches. Additionally, the use of virtual memory blocks that are not continuous on physical memory blocks, Figure 2.4, allows programs to store data anywhere within system boundaries, which is particularly important in multicore systems that can execute programs with different criticality levels that operate on varying amounts of data. The hypervisor can also assign a different color to each guest by choosing the color index. The number and size of colors are hardware-dependent [25].

Cache locking is another technique that can be used to minimize interference in shared caches. It involves locking specific cache lines or sets, preventing them from being evicted and ensuring that they are available for a specific core or task. This technique can be used with cache coloring to further reduce interference and improve predictability. There are three ways to lock cached content: (i) through an instruction that fetches and locks a given cache line, (ii) defining the lock status of every cache way for each CPU in the system, figure 2.5a (called lockdown by the master [36]), and (iii) setting all cache lines fetched as locked until an unlock operation is performed, figure 2.5b.

Address Set	way 0	way 1	way 2	way n
index 0	locked			
index 1	locked			
index 2	locked			
index n	locked			

(a) Cache way locking.

Address Set	way 0	way 1	way 2	way n
index 0				
index 1	locked			
index 2				
index n				

(b) Cache line locking.

Figure 2.5: Cache Locking by way and by line.

Cache locking is a useful mechanism for improving the performance of real-time applications and easing WCET estimation. Consider a dual-core platform with a 2-way set-associative cache and a cache controller that implements a lockdown by master mechanism. It is possible to set the hardware to unlock a specific cache-way for each CPU, allowing a task running on CPU1 to deterministically allocate blocks on way 1 that can never be evicted by a CPU2 task accessing way 2 [35]. This provides more predictable and controllable access to shared caches.

Memory Bandwidth Partitioning

Memory bandwidth reservation is implemented to mitigate interference in multicore systems by allocating a specific amount of memory bandwidth to a domain or group of domains. The main objective of this technique is to improve memory access patterns and reduce contention for memory resources. It ensures that each domain has a guaranteed amount of memory bandwidth (needed to execute their workloads, even when other domains are also accessing memory).

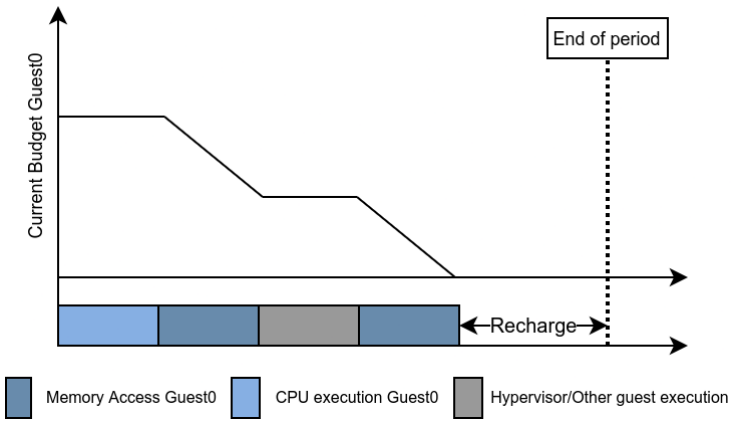


Figure 2.6: Memory Reservation Scheduling. Adapted from [25].

Different approaches have been proposed to implement memory bandwidth reservation, such as time-division multiplexing, space-division multiplexing, priority-based multiplexing, or hybrid techniques. These approaches can be implemented either statically, based on a priori knowledge of the workloads and system characteristics, or dynamically, based on the current state of the system. This technique has been found to be useful in real-time systems, where domains have strict real-time requirements, as it helps to ensure that these domains have the memory bandwidth necessary to meet their deadlines.

For instance, memory bandwidth reservation can be implemented at the hypervisor level and provides each domain with a budget of memory access that is replenished periodically (as shown in Figure 2.6). The budget and period values are statically configured for each domain by the system. This software-based memory throttling mechanism and the analytic solutions for computing throttling parameters help to ensure the schedulability of critical tasks and minimize the performance impact on cores through the configuration of budget (number of memory accesses allowed for each domain in each period) and period values [37]. Also, this technique allows the mitigation of the interference incurred by the different domains due to the memory contention, which becomes limited by the memory budget per CPU or VM, regardless of other domains behavior [2].

2.4 Related Work

The goal of the dissertation is to implement interference mitigation techniques on the Bao hypervisor [4]. The dissertation will likely include a background analysis of existing hypervisors, such as Xen [38], that have similar features. Xen is a type-1 hypervisor that uses a privileged VM called Dom0 to manage non-privileged VMs (DomUs) and interact with peripherals. The dissertation focus on understanding how these existing techniques are implemented, and potentially adapting or improving upon them for use in the Bao hypervisor.

2.4.1 Hypervisors

The Jailhouse hypervisor [39] is a static partitioning hypervisor that utilizes a minimal software layer and hardware-assisted virtualization techniques to statically isolate all platform resources and assign them to a single VM instance. Its unique approach is to assign each virtual core directly to a physical CPU in a 1:1 ratio, which reduces the size and complexity of the system by eliminating the need for a scheduler. Although static partitioning guarantees isolation and real-time performance, it can be challenged by the need for efficient resource utilization. Despite its design philosophy, Jailhouse still has some limitations similar to other hypervisors, such as its reliance on a privileged Linux VM to initialize the system and manage other VMs.

Bao hypervisor, a static partitioning hypervisor that focuses on providing high safety and security guarantees. Despite having a similar architecture to Jailhouse, Bao does not depend on external sources

(except the firmware to perform low-level platform management). It is composed of a thin layer of privileged software that separates the hardware using Instruction Set Architecture (ISA) virtualization extensions, and includes hardware-based solutions like cache coloring that mitigates interference between guests. Furthermore, this can be exploited by a malicious VM to implement DoS attacks, increasing their consumption of a shared resource [20], or accessing data from other VMs through the implicit timing side-channels.

2.4.2 Mitigation Mechanisms

The literature on techniques to control and analyze contention is too vast to be fully covered within the scope of this text. Therefore, a selection of the most relevant works is presented below. Yun et al. [30] proposed one of the first methods to regulate memory bandwidth with their implementation of a per-core regulator, which uses performance counters to implement a budgeting mechanism on a hypervisor.

MemGuard [30] divides memory bandwidth into two categories: guaranteed and best effort. The guaranteed bandwidth is the minimum service rate that the DRAM system can provide, while the additional available bandwidth is classified as best effort and cannot be guaranteed by the system. To ensure temporal isolation, this mechanism focuses on the guaranteed component and effectively utilizes the guaranteed memory bandwidth based on each core's usage estimation. Once the guaranteed bandwidth of each core is satisfied, the best-effort bandwidth can be utilized to improve system throughput.

A Survey on Cache Management Mechanisms by Gracioli et al. [35] and the methods proposed by Yun et al. [40] and Kim et al. [41], suggested several techniques to improve the predictability of cache memories, which reduce contention through bank-aware memory allocations. The survey of cache management techniques for real-time embedded systems notes that one of the main factors contributing to unpredictability in multicore processors is the cache memory hierarchy. CPU caches are hardware elements that are largely invisible to programmers and rely on the spatial and temporal proximity of memory accesses to improve application execution time. They use heuristics to replace outdated, non-referenced entries with data that is more likely to be accessed soon. However, the heuristic behavior of a cache means that, based on the system's history, memory accesses made at the same location during a task's execution may or may not result in a cache hit [35].

PALLOC suggested by Yun et al. [40] include the use of a DRAM bank-aware memory allocator to reduce contention through bank-aware memory allocations. This allocator takes advantage of the page-based virtual memory system to assign memory pages to specific banks for each application. With PALLOC, we can effectively improve isolation on COTS multicore platforms without requiring additional hardware support by dynamically partitioning banks to prevent sharing among cores. A system designer can also use PALLOC to flexibly divide DRAM banks to enhance performance isolation on the multicore platform. For example, private DRAM banks can be assigned to each virtual scheduling division and to each core, eliminating bank sharing between cores without requiring hardware modifications. Smart bank assignment schemes can significantly reduce the limited space issue due to PALLOC's ability to dynamically adjust its assignments at runtime. DRAM is used as the main memory in modern COTS-based systems to meet high

performance and capacity demands. However, the DRAM system has multiple resources, so the access time varies significantly based on the requested address. In addition, memory requests are scheduled by an on-chip, out-of-order memory controller based on the First-Ready First-Come First-Serve (FR-FCFS) policy, where requests that arrived earlier may be serviced later than ones that arrived later if the memory system is not ready to service the former. This technique suggests a white-box strategy for limiting memory interference by explicitly considering the timing characteristics of major DRAM system resources, including the re-ordering effect of FR-FCFS and its timing constraints, to obtain a tight upper bound on the worst-case memory interference delay for a task when it executes in parallel with other tasks.

E-warp by Sohal et al. [42] have implemented a framework for analyzing the memory demand and predicting the timing of real-time workloads on CPUs and hardware accelerators. This framework makes accurate predictions about the temporal behavior of workloads running on CPUs and accelerators by examining the memory demand of applications using a profile-driven approach and performing saturation-aware system consolidation.

Modica et al. [2] have also implemented techniques to address the problem of providing spatial and temporal isolation between execution domains on a hypervisor running on an Arm multicore platform. They specify the memory layout for each domain, which is organized into memory regions and supports coloring through a segmentation scheme that reserves an area of physical memory for each region and splits it into maps.

COLORIS the memory management framework called COLORIS addresses the problem of interference between shared resources by providing support for both static and dynamic cache partitioning, using page coloring. This framework monitors the cache miss rates of running applications and repartitions the cache to prevent miss rates from exceeding application-specific ranges. COLORIS consists of two main components: a Page Color Manager and a Color-aware Page Allocator. The Color-aware Page Allocator allocates page frames of specific colors, and the Page Color Manager is responsible for assigning initial page colors to processes, monitoring process cache usage metrics, and adjusting color assignments based on system-specific objectives such as fairness, QoS, or performance [43].

Concluding, there have also been several efforts to design custom memory controllers to enhance predictability and improve system performance [44, 45], but these designs are not present in COTS platforms, limiting their adoption. In contrast, the MPAM specification has the potential to be present in all Arm platforms and therefore billions of devices. However, only one previous study has examined the effects of the MPAM bandwidth component [46]. This work outlines possible instantiations of the specification at the DRAM memory controller level, highlights many points where it is underspecified, and analyzes memory contention with different design alternatives for MPAM bandwidth partitioning.

3. Platform and tools

Arm processors have significantly improved their architectures for embedded systems, with a focus on providing processors with competitive processing power, and low cost and power consumption [2]. This section discusses the hardware platform and tools used in the design, development, and testing of the proposed work. It first provides an overview of some Bao-supported Armv8-A platforms and then explains why the Zynq UltraScale+ MPSoC ZCU104 was chosen as the target platform to deploy the first mechanism (safe_mem). Additionally, the API used to manage the MPAM registers will specify the configurations necessary to integrate the MPAM processor extension. The scope and explanation of the benchmarks used to confirm the effectiveness of the applied techniques are also presented in the last section.

3.1 Platforms

The implementation of the safe_mem mechanism requires the utilization of PMU and its registers for its implementation. This section outlines and compares different platforms endowed with Armv8-A CPUs (e.g. Cortex-A53), analyzing its advantages and drawbacks.

3.1.1 Armv8-A Platforms

Zynq UltraScale+ MPSoC ZCU104

The ZCU104 board, equipped with a Zynq UltraScale+ MPSoC by Xilinx, includes an Arm Cortex-A53 (quad-core), which is low-power and implements the Armv8-A architecture, running at 1.2 GHz, a Generic Interrupt Controller (GIC)-400 (GICv2) featuring four list registers, and an Memory Management Unit (MMU)-500 (SMMUv2). Cores each with a 32KB L1 instruction and data cache and share an L2 1MB unified cache [47].

NVIDIA Jetson TX2

This board is composed of two CPU clusters (six cores total) in a coherent multi-processor configuration – NVIDIA Denver 2 (dual-core) Processor; Arm Cortex-A57 MPCore (Quad-Core) Processor. Denver 2 and Cortex-A57 CPU clusters support Armv8 architecture, implements GIC-400 (GICv2), and are connected by

a high-performance coherent interconnect fabric designed by NVIDIA, enabling the simultaneous operation of both CPU clusters. Further, both cores in the Denver 2 processor are identical implementations of the Armv8 architecture with NVIDIA optimizations, each includes 128KB Instruction (I-cache) and 64KB Data (D-cache) Level 1 cache. Also, they share a 2MB L2 cache. On the other hand, all four cores in the ARM Cortex-A57 are identical implementations of the Armv8 architecture. Each one includes 48KB Instruction (I-cache) and 32KB Data (D-cache) Level 1 cache. Also, a 2MB L2 cache is shared by the cores of this cluster.

NXP i.MX 8MQuad

The board NXP i.MX 8MQuad features a four-core Cortex-A53 Accelerated Processing Unit (APU) cluster at operating frequencies of 1.5 GHz, implements GIC-500 (GICv3), and a general purpose Cortex-M4 microcontroller for low-power processing, operating at 266 MHz. Each core of the Cortex-A53 has a data and instructions cache for level 1 of 32 KB in size, and for level 2, a shared cache of 1 MB. Further, the Cortex-M4 has a total of 32KB L1 cache size, 16KB for instruction, and another 16KB for data.

3.1.2 Platform Selection

This section presents an analysis of the characteristics of each Bao-supported development platform that is based on the type and number of A cores, the memory cache size (L1 and L2), and frequency of operation, as shown in table 3.1. The NVIDIA Jetson TX2 is the recommended platform for the implementation of the mechanisms proposed in this dissertation due to its larger cache sizes and CPU operating frequencies. However, due to the unavailability of this NVIDIA board (a.k.a. silicon shortage), the Zynq UltraScale+ MPSoC ZCU104 was used instead.

Development Board	# of Cores	Core Type (Arm Cortex-A)	Cache Size		Operation Frequency
ZCU104	4	Cortex-A53	L1(I/D): 32KB	L2: 1MB	1.2 GHz
NVIDIA Jetson TX2	4	Cortex-A57	L1(I/D): 48KB/32KB	L2: 2MB	2GHz
NXP i.MX 8MQuad	4	Cortex-A53	L1(I/D): 32KB/32KB	L2: 2MB	1.5 GHz

Table 3.1: Platform Comparison.

3.2 Tools

The second goal of this dissertation involves implementing the MPAM extension, emulating the hardware using an FVP model, and debugging the simulation model using Arm DS. This is necessary because,

at the time of writing, no silicon supports the MPAM extension.

3.2.1 Arm Fast Models

Arm Fast Models are accurate, flexible programmer's view models of Arm IP, allowing the development of software such as drivers, firmware, OS, and applications before silicon availability. They enable simulation control, including profiling, debuging, and tracing. Also, these models are included in the larger System on Chip (SoC) design process by being exported to SystemC and TLM 2.0. An Instruction Set Simulator is not a complete virtual prototype of a system. It also includes fast and accurate models of processors, subsystems, or systems. Further, it provides APIs for debuging (CADI) that allows full control of system execution, an interface to Arm DS-5 (section 3.2.2) and other debug tools. It also includes visualization, file system access, and peripherals from I/O. The Arm Fast Models simulate the Versatile Express Architecture Envelope Models (AEM) VA hardware platform that implements all architectural features in the Armv8-A instruction set. The model included a four Arm A cores with virtualization extension, custom shared size cache, bandwidth, memories, and configurable MPAM parameters, emulating its behavior.

3.2.2 Arm Development Studio

Fast Models offers fixed versions, known as FVPs, which provide programmers with a fully functional model of an entire Arm system. These FVPs can be easily downloaded, individually licensed, and imported into Development Studio for simplicity of use. The model chosen to run in this tool was based on its ability to emulate the Armv8-A architecture and MPAM extension behavior. Therefore, the Arm base RevC AEMv8A FVP was selected due to its ability to emulate a generic Armv8-A 64-bit hardware platform with MPAM support [48].

FVP Model Configuration

Fixed Virtual Platforms are pre-configured, functionally accurate simulations of system configurations. This configuration takes into account the implementation of the MPAM extension, so some parameters, among others, must be enabled and assigned, when running the simulation model. The different parameters related to the implementation of MPAM extension are:

- **cluster0.has_mpam=2** : enable MPAM registers and associated functionality (FEAT_MPAM);
- **cluster0.l3cache-has_mpam=1** : enable MPAM for L3 cache;
- **cluster0.l3cache-mpamf_base=0x100000000** : base address for memory page of MPAM memory mapped registers.

Also, it is necessary to configure and assign some parameters about the cache of the system to use the MPAM mechanism for cache partitioning. The parameters assigned for this purpose are listed below:

- **cache_state_modelled=true** : enable d-cache and i-cache state for all components;
- **cluster0.l3cache-size=1024** : indicate the size of cache in bytes (1MB);
- **cluster0.l3cache-mpamf.cmax_width_ns=16** : indicate the maximum cache capacity usage for each partition;
- **cluster0.l3cache-mpamf.cpbm_width_ns=32** : indicate the number of portions available of system cache;

3.2.3 Test Benchmarks

Synthetic benchmark

This benchmark is memory intensive and uses two bare-metal applications running in separate VMs, with the primary objective of stressing the memory, more explicitly LLC. One of these applications runs on a single CPU and is responsible for verifying and displaying all of the selected PMU event values of each cache memory access after a pre-configured number of samples. The other application, run by the other three CPUs, is responsible for inducing interference in the system. Further, for the results given by the benchmark to be more reliable, a cache warming function is used to fill the memory to increase the efficiency of cache access and decrease the occurrence of cache misses.

Last but not least, it should be mentioned that a memory-intensive application running in a VM, with three CPUs, would cause more interference in LLC than on the system bus. Also, the system takes fewer time cycles in the memory bus since CPUs from the same system domain are accessing the main memory at a time. Therefore it is essential to select the working set for each baremetal application. The application responsible for monitoring the performance of the memory-accessing application should be larger than the cache L2 size because it is the only way to access the LLC. On the other hand, if its working set is too large, it will lead to migrating the interference to the system bus because the CPUs of the other application are also accessing the memory to create interference. In addition, the working set chosen for the interference application must cover the whole LLC.

MiBench

As outlined in section 5.1, to evaluate the performance overhead and interference between virtual machines, we use the MiBench benchmark suite [49]. MiBench is a collection of 35 benchmarks that are divided into six categories, each targeting a specific area of the embedded market such as automotive and industrial control, consumer devices, office automation, networking, security, and telecommunications. We focus our evaluation on the automotive and industrial control category as it is one of the main application domains targeted by Bao. This category includes three of the more memory-intensive benchmarks, which are more susceptible to interference due to cache and memory contention [50] (qsort, susan corners, and susan edges). Additionally, the tests consist of memory accesses and their impact on the system's overall performance according to these memory-intensive benchmarks.

4. Design and Implementation

The previous chapter introduced the main concepts related to this dissertation. This chapter provides an overview of the mechanisms implemented in the Bao hypervisor and offers background information on its specifications and characteristics within the system. This chapter also specifies the proposed safe_mem mechanism (Section 4.1) and the MPAM API (Section 4.2), including the design and all specifications for both implementations in the hypervisor.

4.1 Memory Throtlling - 'safe_mem'

The term "throttling" refers to methods for managing and allocating shared resources to different domains of a system. Also, through the hypervisor, the system is partitioned and assigned to different guests, but there are still shared resources, particularly DRAM, that are used by all guests. As a result, DRAM can be a source of contention, as tasks running concurrently on different CPUs compete for access, leading to longer execution times. Throttling techniques use the PMU to monitor microarchitectural events and manage workload requests so that they do not exceed a well-defined threshold, ensuring the proper operation of the system. In real-time systems, the effectiveness of throttling depends on time constraints, the hardware used, and software integration. The memory bandwidth reservation proposed in this dissertation relies on a budget-base approach that leverages two hardware components: (i) the aarch64 generic timer and (ii) the PMU. It sets a budget value for each guest and an sample period to monitor DRAM accesses by each guest per period, limiting them to the defined budget per period.

4.1.1 Aarch64 Timer

This topic describes the Aarch64 generic timer which provides a standardized timer framework for Arm cores, including the System counter and a set of per-core timers, as shown in table 4.1. System Counter is an always-on device that provides a fixed frequency incrementing system count, designed by the SoC implementer, requiring initialization when a system boots up. Additionally, its value is broadcast to every core in the system at a frequency of between 1MHz and 50MHz, with a width of between 56 bits and 64 bits, giving them all a common perception of timing. The timer can be also configured to generate an interrupt [51]. The Cortex-A53 processor provides to each core a set of comparator timer registers relative

to Arm-v8 that compares its value against the commonly broadcast system count supplied to each core by the system counter.

Timer	**Register prefix**	**Exception Level - 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

Table 4.1: Timers.

The following registers can program each timer, register prefix, given by table 4.1, to control and monitor the timer.

- **CNTFRQ_EL0**: report the frequency of the system count;
- **<timer>_CTL_EL<x>**: counter timer control register;
- **<timer>_CVAL_EL<x>**: counter timer comparator value register;
- **<timer>_TVAL_EL<x>**: counter-timer timer value register.

The timer used for the implementation of this mechanism is the "Non-secure EL2 physical timer." The choice was made because the implementation of this mechanism is established at the EL2 layer, the hypervisor layer. It is worth noting that specific timer registers for configuring/generating an interrupt for a fixed time interval, controlled through the CTL register, are used to enable an interrupt for each core in this implementation. The timer is configured through the TVAL register, which reads the current system count internally, adds the specified value, and then populates the CVAL register, triggering an interrupt when the system count reaches that value. In summary, this timer is used to trigger an interrupt at the end of each period, resetting its predefined value. Its role in controlling the memory usage of each guest per period with the help of the PMU is discussed in topic 4.1.3.

4.1.2 Performance Monitoring Unit

The Cortex-A53 processor includes performance monitors that implement the Arm PMUv3 architecture. The PMU is a functional unit that records hardware-related performance monitoring events and provides relevant information about the behavior of the processor and its memory system during runtime.

It offers one cycle counter and six event counters (Figure 4.1), each of which allows for the concurrent monitoring of available events in the processor [47].

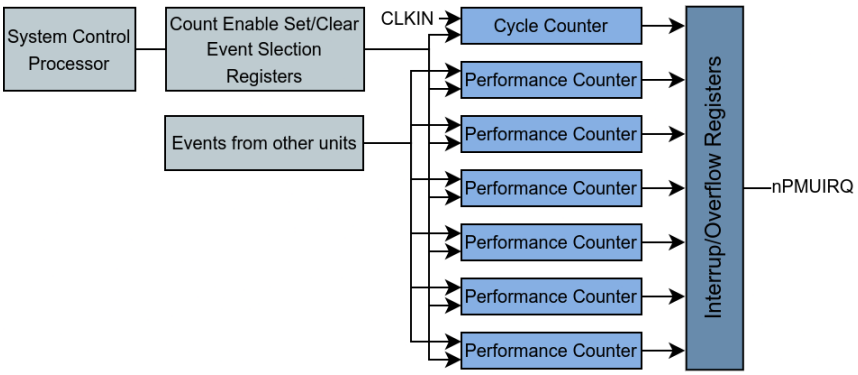


Figure 4.1: PMU Block Diagram. Adapted from [47].

Furthermore, regarding the purpose of the PMU, in the following table are some of the controllable events that can be used, together with the timer (Section 4.1.1), to implement the safe_mem mechanism:

Event	Event mnemonic	Description
Memory Access	MEM_ACCESS	counts memory accesses due to load/store instructions
Bus Access	BUS_ACCESS	counts data transferred between the core and the SCU
L2 Data cache refill	L2D_CACHE_REFILL	counts any cacheable transaction from L1 which causes data to be read from outside the core
L2 Data cache access	L2D_CACHE	counts any transaction and any write-back from the L1 to the L2

Table 4.2: PMU Event Counters.

The PMU counters and associated control registers can be accessed in the AArch64, the 64-bit execution state of the ARMv8 ISA, using the MRS and MSR instructions. The table 4.2 summarizes the Cortex-A53 PMU registers used to count a desired event in the AArch64 execution state. These registers can be used to configure and implement the PMU to count a specific event and monitor the memory usage per period by each core.

- **MDCR_EL2**: Enable event counters;
- **PMCNTENSET_EL0**: Enable cycle register;
- **PMSELR_EL0**: Selects the current event counter;
- **PMEVTYPER<n>_EL0**: Configures event counter;
- **PMINTENSET_EL1**: Enable counter overflow interrupt request;
- **PMOVSCLR_EL0**: Clear counter overflow status.

Additionally, the BUS_ACCESS event of the Cortex-A53 processor was chosen for use in this mechanism as it counts any data transfer from or to the System Control Unit (SCU). The reason for this selection is that the goal of this mechanism is to mitigate DRAM contention, and this event has the greatest impact on interference control at the DRAM level. It also significantly reduces the impact of interference, improving system performance. The next section explains the implementation of this event in relation to the purpose and relevance of each register within this mechanism.

4.1.3 Integration of safe_mem in Bao Hypervisor

The main concepts related to this mechanism have been discussed in previous sections, including the design of the technique to control the system's shared resources through the timer registers of each CPU and the PMU to limit the memory usage. In this section, the implementation and configuration of the Aarch64 timer and PMU are first outlined. The usage of the timer and PMU, as well as their interrupt capabilities within the mechanism, are then explained in detail. These components are used to reduce contention on the DRAM, shared among all cores of the system.

The aarch64 timer is particularly important in this mechanism as it periodically triggers the system with a predefined time interval. The initialization encompasses receiving the period value, defined by the user, in microseconds (Listing 4.1). The required number of counts to trigger the timer is calculated based on this period and the system frequency obtained using the *timer_arch_get_system_frequency* function. The timer counter is then set through the configuration of its registers. Finally, the timer is enabled, its interrupt is enabled, and the function returns the calculated count number.

```
uint64_t timer_arch_init(uint64_t period)
{
  uint64_t frequency;
  uint64_t count_value;

  //set timer counter
  frequency = timer_arch_get_system_frequency();
  count_value = (period*frequency)/1000000;

  timer_arch_set_counter(count_value);
  timer_arch_enable();

  return count_value;
}
```

Listing 4.1: Sampling period timer initialization.

Furthermore, the PMU is responsible for counting the number of occurrences of a selected event from the options available on this CPU. To monitor a specific event, it is necessary to select and configure a

PMU counter and implement the desired events that may be useful for the purpose of the mechanism. The first step is to enable the PMU at EL2 and check the number of implemented counters, as well as defining the number of event counters accessible from EL3, EL2, EL1, and EL0 (see Listing 4.2).

```
#define PMCR_EL0_N_POS     (11)
#define PMCR_EL0_N_MASK    (0x1F << PMCR_EL0_N_POS)

#define MDCR_EL2_HPME    (1 << 7)
#define MDCR_EL2_HPMN_MASK   (0x1F)

#define PMU_N_CNTR_GIVEN     1

static size_t events_array[] = {DATA_MEMORY_ACCESS, L2D_CACHE_ACCESS,
    BUS_ACCESS, L2D_CACHE_REFILL};

void pmu_enable(void)
{
    uint32_t pmcr = MRS(PMCR_EL0);
    uint64_t mdcr = MRS(MDCR_EL2);

    cpu.implemented_event_counters = ((pmcr & PMCR_EL0_N_MASK) >>
    PMCR_EL0_N_POS);

    //Enable the pmu at EL2
    mdcr &= ~MDCR_EL2_HPMN_MASK;
    mdcr |= MDCR_EL2_HPME + (PMU_N_CNTR_GIVEN);

    MSR(MDCR_EL2, mdcr);
}
```

Listing 4.2: PMU configuration for EL2.

Additionally, one of the available counters of the PMU can be enabled to count the desired PMU event. The overflow value for the counter is also defined in Listing 4.3. Finally, the chosen event (as shown in Listing 4.4) can be monitored using both the PMU and timer mechanisms.

```
static inline void pmu_cntr_set(size_t counter, unsigned long value)
{
    uint64_t pmselr = MRS(PMSELR_EL0);
    pmselr = bit_insert(pmselr, counter, 0, 5);
    MSR(PMSELR_EL0, pmselr);

    value = UINT32_MAX - value;
    MSR(PMXEVCNTR_EL0, value);//Define overflow value for pmu counter
}
```

Listing 4.3: PMU event overflow value definition.

```
static inline void pmu_set_evtyper(size_t counter, size_t event)
{
    uint64_t pmselr = MRS(PMSELR_EL0);
    pmselr = bit_insert(pmselr, counter, 0, 5);
    MSR(PMSELR_EL0, pmselr);

    uint64_t pmxevtyper = MRS(PMXEVTYPER_EL0);
    pmxevtyper = bit_clear(pmxevtyper,PMEVTYPER_P);
    pmxevtyper = bit_clear(pmxevtyper,PMEVTYPER_U);
    pmxevtyper = bit_clear(pmxevtyper,PMEVTYPER_NSK);
    pmxevtyper = bit_clear(pmxevtyper,PMEVTYPER_NSU);
    pmxevtyper = bit_clear(pmxevtyper,PMEVTYPER_NSH);
    pmxevtyper = bit_clear(pmxevtyper,PMEVTYPER_M);
    pmxevtyper = bit_clear(pmxevtyper,PMEVTYPER_MT);
    pmxevtyper = bit_clear(pmxevtyper,PMEVTYPER_SH);
    pmxevtyper = bit_insert(pmxevtyper, events_array[event], 0, 10); //Set
    event to count for pmu event counter */

    MSR(PMXEVTYPER_EL0, pmxevtyper);
}
```

Listing 4.4: PMU event selection.

Also, the PMU event counter allocation function for the mechanism sets a bitmap of implemented event counters, allocating the desired number of counters to the different events of PMU, as shown in Listing 4.5, ensuring scalability through Bao hypervisor's modular architecture.

```
#define PMU_N_CNTR_GIVEN 1
cpu.implemented_event_counters = 0; //init as 0

uint64_t pmu_cntr_alloc()
{
    uint32_t index = PMU_N_CNTR_GIVEN;

    for (int __bit = bitmap_get(cpu.events_bitmap, index); index < cpu.
    implemented_event_counters;
                        __bit = bitmap_get(cpu.events_bitmap, ++index))
        if(!__bit)
            break;

    if(index == cpu.implemented_event_counters)
```

```
14          return ERROR_NO_MORE_EVENT_COUNTERS;

16      bitmap_set(cpu.events_bitmap, index);
        return index;
18  }
```

Listing 4.5: PMU event counter allocation.

The function, shown in Listing 4.6, configures the PMU to count the desired event. It sets the limit of event occurrences before an interrupt is triggered and assigns the function to be called when an interrupt is triggered. It also clears the counter overflow status and enables the CPU and selected counter interrupts. Finally, it enables the PMU and a specific counter using the abstraction function *events_enable*, shown in Listing 4.2.

```
void safe_mem_events_init(events_enum event, unsigned long budget,
    irq_handler_t handler)
2 {
  if((cpu.safe_mem.counter_id = events_cntr_alloc()) ==
   ERROR_NO_MORE_EVENT_COUNTERS)
4  {
    ERROR("No more event counters!");
6  }
  events_set_evtyper(cpu.safe_mem.counter_id, event);
8  events_cntr_set(cpu.safe_mem.counter_id, budget);
  events_cntr_set_irq_callback(handler, cpu.safe_mem.counter_id);
10 events_clear_cntr_ovs(cpu.safe_mem.counter_id);
  events_interrupt_enable(cpu.id);
12 events_cntr_irq_enable(cpu.safe_mem.counter_id);
  events_enable();
14 events_cntr_enable(cpu.safe_mem.counter_id);
}
```

Listing 4.6: PMU event counter init.

After discussing the initialization functions for the two resources used in the mechanism, we describe the functions executed when an interrupt is triggered by either the timer or the PMU. The PMU event overflow function is responsible for putting the CPU that has exceeded the defined limit for memory accesses per period into idle mode until the end of the period, as described in Listing 4.7.

```
1 void safe_mem_process_overflow(void)
{
3  events_clear_cntr_ovs(cpu.safe_mem.counter_id);
  events_cntr_disable(cpu.safe_mem.counter_id);
5  events_cntr_irq_disable(cpu.safe_mem.counter_id);
```

```
7   cpu.safe_mem.throttled = true;

9   cpu_idle(); //throttle core
}
```

Listing 4.7: PMU callback function.

The timer callback function (Listing 4.8) is invoked when the timer reaches the end of its period, rescheduling the interrupt, resetting the limit value for memory accesses and verify the throttle state of each CPU, removing the throttled CPUs from idle. Finally, it enables the event counter and its interrupt, starting the timer.

```
void safe_mem_period_timer_callback(irqid_t int_id)
2 {
  timer_disable();
4 events_cntr_disable(cpu.safe_mem.counter_id);

6 //timer
  timer_reschedule_interrupt(cpu.safe_mem.period_counts);
8
  //events
10 events_cntr_set(cpu.safe_mem.counter_id, cpu.safe_mem.budget);
  if(cpu.safe_mem.throttled)
12 {
    events_cntr_irq_enable(cpu.safe_mem.counter_id);
14  cpu.safe_mem.throttled = false;
  }
16 events_cntr_enable(cpu.safe_mem.counter_id);
  timer_enable();
18 }
```

Listing 4.8: Timer callback function.

In conclusion, a flowchart of the mechanism's initialization is presented after detailing its key aspects.

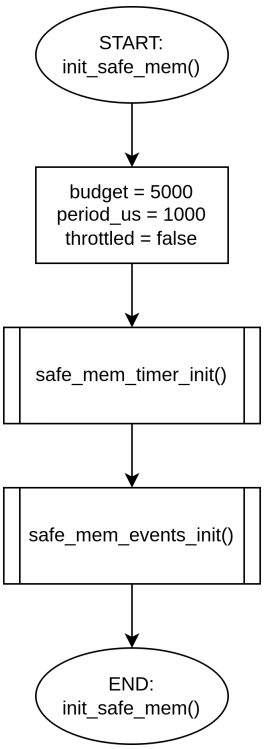


Figure 4.2: safe_mem Initialization Flowchart.

To summarize, this mechanism sets the system's period and the budget for each CPU and then sets the boolean variable "throttled" to false to indicate that all CPUs are active. It then uses the two initialization functions described above to control memory accesses per period using the timer and PMU. This helps to reserve bandwidth for each running CPU, improving system predictability and performance.

4.2 MPAM - Arm Extension

Predictable memory access is a crucial issue for modern heterogeneous platforms. As computing platforms with many processor cores and hardware accelerators become increasingly common to meet the demands of demanding workloads such as those in autonomous driving applications, the finite number of memory system partitions and existing software-based mitigation strategies become limitations. Arm's MPAM specification addresses this issue by offering several memory-access restriction schemes [46].

The MPAM extension is an architectural approach to resource contention avoidance that provides workload identification of memory traffic throughout the system and standard monitoring and control interfaces for workload performance and resource allocation such as cache capacity and memory bandwidth [52]. MPAM identifiers can be attached to memory system requests from CPUs or to device traffic going through the System Memory Management Unit (SMMU) [21]. This extension is designed for shared-memory computer systems that run multiple applications or VMs concurrently. It uses two approaches: resource

partitioning and resource monitoring, which work together to allocate the performance-critical resources of the memory system. This approach divides cache and bandwidth into partitions, allowing for the replacement of software mechanisms such as cache coloring and memory throttling. In this work, only the resource partitioning approach was used due to the unavailability of the platform model used to emulate the system.

4.2.1 Memory-System Resource Partitioning

Memory-system resource partitioning, introduced in Armv8-A, enables the partitioning of Memory System Components (MSCs) shared among different applications and VMs. The specification states that various components such as caches, interconnects, and DRAM memory controllers may support this extension.

According to the MPAM specification, different sources of memory transactions are identified by partition identifiers (PARTIDs), which are used to determine the partitioning of memory resources, and controls its usage in a shared memory system by multiple VMs, OSs, and applications. Resource partitioning is based on the PARTID, which identifies a VM. Resource management is then achieved by the propagation of the PARTID throughout the system. In a system that implements MPAM, when a new memory request is received, the current resource usage is compared to the control requirements set for the PARTID associated with the request [46].

The MPAM extension specification also introduces several mechanisms to improve memory access predictability and isolation. It provides two main categories of techniques: those related to caches and those related to memory bandwidth regulation. The first category includes cache-portion partitioning, which allows for the allocation of portions of the system cache to partitions, and cache maximum-capacity partitioning, which sets a limit on the storage used by a PARTID. The second category includes memory-access regulation techniques: memory-bandwidth minimum-maximum partitioning, priority partitioning, and memory-bandwidth portion partitioning. These three techniques manage memory bandwidth by configuring the maximum, minimum, and portion of total bandwidth assigned to a PARTID, as shown in Figure 4.3, as well as the priority of the PARTID in case of contention.

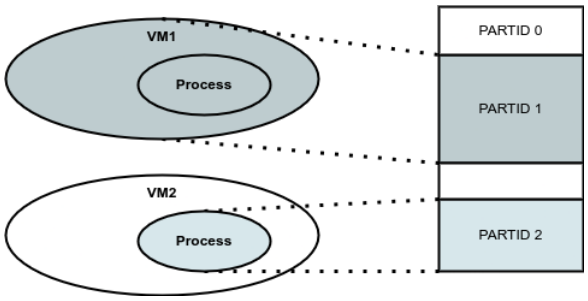


Figure 4.3: System Assignment to MPAM Partitions.

The MPAM specification does not provide a formal definition of bandwidth, but it is typically interpreted as the number of memory transactions in a pre-configured time interval. The size of this interval, called the accounting window, is expressed in microseconds for each PARTID.

Additionally, as mentioned earlier, it assigns VMs and applications through the hypervisor and OSs to system partitions, respectively. To summarize, the PARTID of a request controls the use of each resource and is used within the component to select resource control settings for the component's resource allocation and utilization behavior. All memory-system requests with a given PARTID share the resource control settings for that partition.

4.2.2 Resource Partitioning Control Model

Figure 4.4 illustrates the general design of a resource partitioning controller within an MSC. This model is an example of a resource partitioning model that monitors resource usage by each partition and controls it by comparing the measured value to the control settings for the respective resource partition.

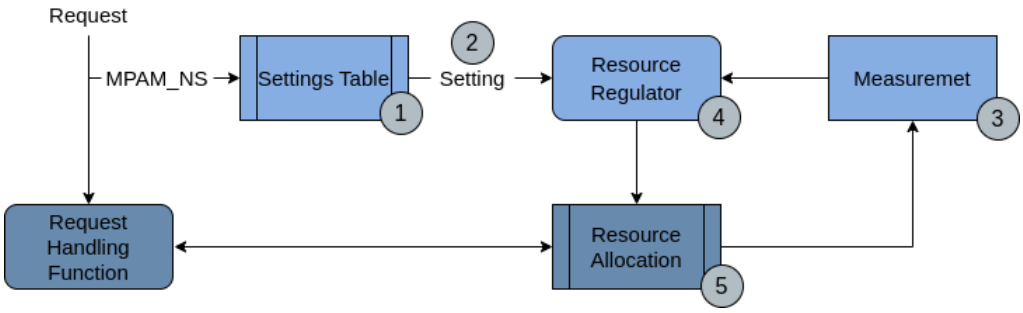


Figure 4.4: MPAM Resource Partitioning Model. Adapted from [52].

As shown, a request arrives from an upstream requester to a component that implements MPAM partitioning control. The handling steps for the request are presented below:

1. To index the settings table of partition control settings, the partition ID and the MPAM NS values, which, respectively, indicate the ID of the partition selected and the physical partition space as Secure or non-secure values of the incoming request (MPAM NS), are used.

2. A Resource Regulator receives the settings from the table entry for that partition, which controls how the resource is segmented.

3. The measured value of the usage of the partitioned resource by the partition must correspond to the resource setting.

4. The measurement feeds back to the Resource Regulator, which is compared with the setting and used to decide the resource allocation.

The items listed above are added to the original memory system when the MPAM extension is enabled. Despite being simple and inexpensive, cache-way partitioning does not allow for fine-grained control due to the limited number of cache ways. Additionally, resources can be constrained if one or more cache ways are allocated to a single partition without sharing.

4.2.3 Integration of MPAM Extension on Bao hypervisor

This section explains how the processor extension is used to control and manage system resources through an API implemented in the Bao hypervisor, describing in detail the various registers, purpose, and design, integration, and implementation of the API.

MPAM System registers

First, it is necessary to configure MPAM, at the exception level where the resource is assigned by the implemented MPAM partition. For example, MPAM can be configured at EL2, the hypervisor level, to manage the different VMs of the system with MPAM version constraints through its API implemented in the hypervisor. To summarize, in a Processing Element (PE), MPAM has system registers to control the generation of partitions by the PE and the assignment of these partitions based on the exception level, version, and configuration of the implemented MPAM registers (Table 4.3) [52].

System Register	Description
MPAM2_EL2	MPAM context for EL2 execution
MPAMIDR_EL1	MPAM identification register (RO)

Table 4.3: MPAM system registers managed.

MPAM External Registers

The behavior of MPAM extension is discovered and configured through MMRs in a MCS. These registers can be found on the MPAM feature page and contain information about the behavior of this extension. There are MMRs for identifying MPAM parameters and options, ID registers, and others for configuring resource controls (Table 4.4) [52]. The correct configuration of cache and bandwidth partitioning can be achieved by modifying certain configuration registers. The previous discussions covered the main concepts related to MPAM, including its functions, registers, their purposes, and how it assigns VMs to different system partitions for cache and bandwidth management. It also provided guidelines for designing and configuring the mechanism for controlling the system through registers assigned to each VM running on the system.

Memory Mapped Register	Description
MPAMCFG_PART_SEL	Select the partition of the system to configure
MPAMCFG_CMAX	Configures the maximum fraction of cache capacity permitted to allocate by a partition
MPAMCFG_CPBM	Configures the cache portions that a partition is allowed to allocate
MPAMF_CCAP_IDR	Indicates the number of fractional bits in MPAMCFG_CMAX (RO)
MPAMF_CPOR_IDR	Indicates the number of bits enable in MPAMCFG_CPBM (RO)
MPAMF_MBW_IDR	Indicates which MPAM bandwidth partitioning features
MPAMCFG_MBW_MIN	Controls the minimum fraction of memory bandwidth that the partition can use
MPAMCFG_MBW_MAX	Controls and limit the maximum fraction of memory bandwidth that the partition can use
MPAMCFG_MBW_PBM	Configures the bandwidth portions that a partition is allowed to allocate

Table 4.4: MPAM MMRs managed.

As MPAM is an extension of Arm processors, which implementation is verified during system initialization, specifically in the VM initialization function: *vm_master_init*, highlighted in Listing 4.9. The function first checks if the platform has this extension, and if so, it implements the pre-configured colors to assign the MPAM partitioning registers. If the platform does not implement the MPAM extension, the cache coloring mechanism with pre-configured colors is used during system configuration.

```
static void vm_master_init(struct vm* vm, const struct vm_config* config,
    vmid_t vm_id)
{
    //(...)
    uint64_t mpam_status;

    mpam_status = MRS(ID_AA64PFR0_EL1);

    if((mpam_status>>40)&0xF) // platform implementes MPAM
    {
        as_init(&vm->as, AS_VM, vm_id, NULL, 0);
    }else
    {
        as_init(&vm->as, AS_VM, vm_id, NULL, config->colors);
    }
    //(...)
}
```

Listing 4.9: MPAM status verification in VM Initialization.

In the context of MPAM, the hypervisor assigns the different partitions to each VM and associates them with all memory system requests originated PE. Thus, this resource partitioning model controls and limits the usage of a resource by the assigned VM. This is accomplished by comparing the usage that is measured to the control settings for that specific VM.

```
void MPAM_cache_partition_config(vmid_t vm_partition, colormap_t cpbm)
{
    mpam_pag.mpam->CFG_PART_SEL = vm_partition;

    mpam_pag.mpam->CFG_CMAX = 0xFFFF; //0.999984741

    mpam_pag.mpam->MPAMCFG_CPBM[0] = cpbm; //cache portions bitmap
}
```

Listing 4.10: MPAM Cache Partitioning Configuration.

The function responsible for partitioning the cache among different VMs, Listing 4.10, first selects the VM to configure. It then assigns the maximum cache capacity usage, in a fixed-point fraction format specified by the user, as a percentage of the total cache capacity for the selected partition. This represents the portion of the total cache capacity that is permitted to be allocated according to the configuration of the FVP model, Section 3.2.2. Finally, it assigns the cache bitmap with all the portions available, according to the configuration of the FVP model, that the selected VM is permitted to allocate. In summary, this function implements the segmentation of the system cache by assigning the configured colors for each partition to the MPAM register responsible for the cache partitioning bitmap (MPAMCFG_CPBM). The value of this register must be in accordance with the platform parameters that set the available portions in the system cache and the maximum cache capacity for each partition.

Regarding the partitioning of memory bandwidth for the different assigned VMs, the configuration function assigns one of the running VMs and sets the maximum and minimum bandwidth usage values for that VM. If the maximum bandwidth value is exceeded during the accounting window, the partition will not use any more memory until the memory bandwidth measurement for that partition falls below that maximum bandwidth value and can compete with other applications, as allowed by other regulation mechanisms. If the maximum bandwidth value is not exceeded, the requests from this partition are preferentially selected to be served. The function also sets the number of implemented bits in the bandwidth allocation fields according to the platform configuration, setting the bandwidth portions available for the selected VM (Listing 4.11). Both the maximum and minimum bandwidth values are assigned to the system partitions in accordance with the MPAM manual, expressed in a fixed-point fraction, specified as a percentage of the total bandwidth of the system.

```
void MPAM_bw_partition_config(vmid_t vm_partition, colormap_t bwpbm)
{
    uint8_t bw_width;
```

```
4
    mpam_pag.mpam->CFG_PART_SEL = vm_partition;
6
    bw_width = mpam_pag.mpam->MBW_IDR & 0x3F; //Number of implemented bits
   in the bandwidth allocation fields: MIN, MAX
8
    mpam_pag.mpam->CFG_MBW_MIN = 0xFF00;
10  mpam_pag.mpam->CFG_MBW_MAX = 0x8000FF00;

12  mpam_pag.mpam->MPAMCFG_MBW_PBM[0] = bwpbm;
}
```

Listing 4.11: MPAM Bandwidth Partitioning Configuration.

After discussing the cache and bandwidth partitioning configuration functions, the setup of the MPAM system registers to enable this mechanism at the correct exception level, through the MPAM2_EL2 register, for managing the different VMs and their system resources is explained. To implement this extension, it is necessary to allocate a virtual memory page, whose base address is specified in the platform's configuration file 3.2.2, for MPAM MMRs to access and assign them to the control settings for cache and bandwidth. This page also includes all the MMRs for the MPAM extension. The diagram 4.5 shows the configuration for this mechanism, including the functions used to assign the registers and to control the partitioning of cache and bandwidth, selected in the hypervisor configuration of the system domains.

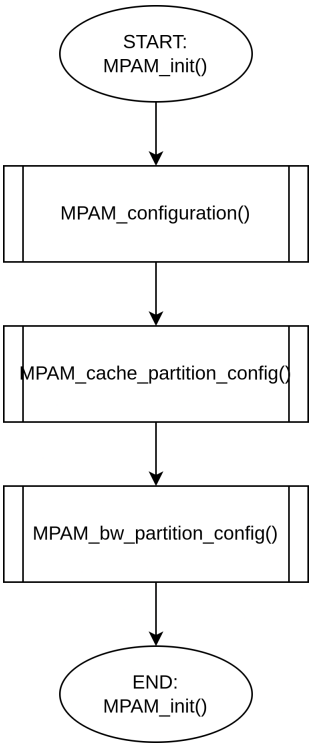


Figure 4.5: MPAM Initialization Flowchart.

The functions presented in the diagram are executed during the initialization of each VM so that every VM running on the system will assign its partitioning control registers for cache and bandwidth according to the configuration. As mentioned, the system only executes these functions if it has the MPAM extension implemented, which is verified through the processor register (ID_AA64PFR0_EL1). This determines whether to use the software mechanisms for cache partitioning and memory bandwidth reservation or the features of the MPAM extension. These features of this processor extension are configured to achieve better system predictability and determinism, controlling and monitoring the cache and bandwidth, with various VMs running concurrently without interfering with each other.

Despite this extension achieving partial isolation on shared resources, it is important to carefully consider every aspect of the design, such as system frequency, memory cache sizes, and a total understanding of the additional MPAM features when implementing the MPAM API. However, to date, chip manufacturers deploying MPAM may want to consider increasing execution predictability, as no platforms have implemented the extension specification yet. While the MPAM standard provides a high-level description of the regulating mechanisms, it allows for a lot of room for interpretation [46]. This can lead to slight differences in the behaviors of these mechanisms, which can result in significantly different worst-case timing performances.

5. Evaluation and Results

5.1 Evaluation and Results

The previous sections of this dissertation outlined the design and development of the memory throttling algorithm *(safe_mem)* and the MPAM API. This section presents the experimental results of the system validation process, which is divided into two main sections: (i) results of the memory throttling algorithm, and (ii) results of the MPAM API implementation. The first section presents the results of the *safe_mem* algorithm, which was evaluated using the MiBench benchmark [49], a small data set representing a lightweight embedded application that simulates a real-world scenario. Further, the second section presents the results of the MPAM implementation, which was evaluated using a synthetic benchmark (described in detail in Section 3.2.3).

5.1.1 Safe Mem Results

The performance of the *safe_mem* mechanism is evaluated in terms of latency relative to a Linux system running without any concurrent interference application (solo). To measure the impact of interference on DRAM access, we use a baremetal interference application that repeatedly accesses a cache matrix with three lines for each CPU, up to a maximum of three CPUs, and a total size of 1M (cache size). This interference is meant to represent the effects of other applications running on the system.

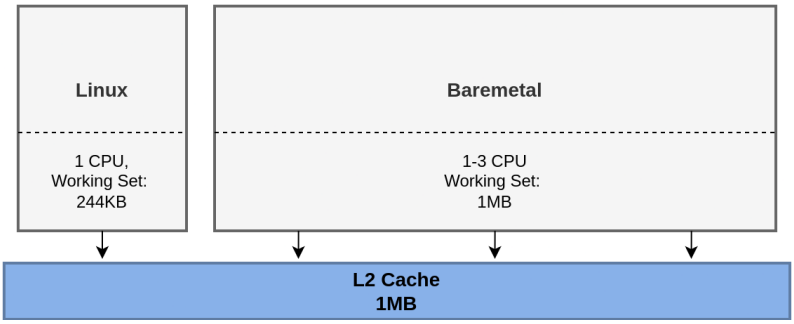


Figure 5.1: Design of the System for Testing safe_mem Mechanism.

Additionally, this evaluation is conducted to determine the PMU event that experiences the smallest performance drop due to the interference induced by the baremetal application. It's important to note

that the budget value is the maximum number of event occurrences for each period. By analyzing the system response with different values of budget and period, we can determine the scenario in which the mechanism is more effective. The design for the tests, depicted in the Figure 5.1, compares the relative performance overhead of a system with one core allocated for Linux (solo) with safe_mem enabled and disabled.

Evaluation

The first set of experiments aims at assessing the performance of the system while running the interference application and Linux with the *safe_mem* algorithm enabled concurrently. These tests show that for the lowest bandwidth value, corresponding to budgets (memory accesses per CPU) of 500, 5000, and 50000, the impact of the *safe_mem* mechanism on the execution time of the benchmark is lower, particularly in the 10ms period. It's worth noting that in this algorithm, budget and bandwidth are both directly proportional to each other and inversely proportional to the period. Therefore, either increasing the budget or decreasing the time period will result in a lesser negative impact of *safe_mem* on system performance. This is supported by the graph in Figure 5.2.

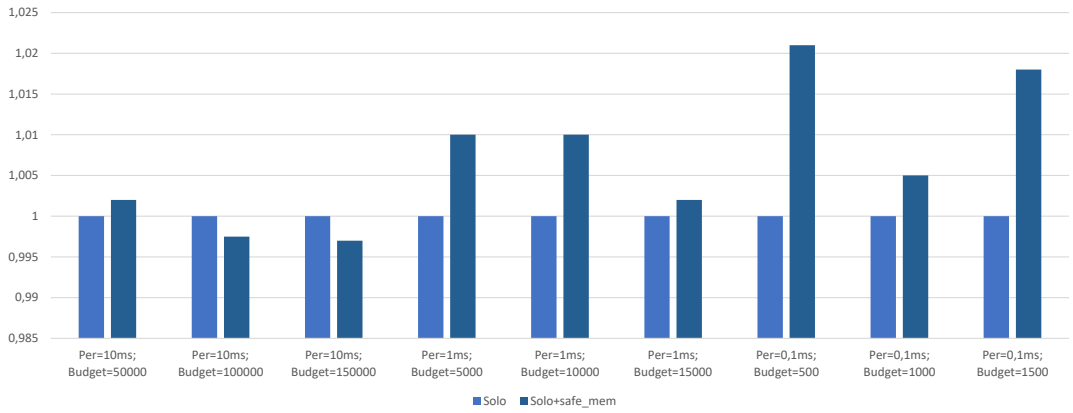


Figure 5.2: Relative Performance Overhead: Solo vs. Solo + safe_mem Algorithm.

Although the individual performance of Linux does show a slight decrease of approximately 2% in system performance with *safe_mem* mechanism enabled for the same bandwidth values, the mechanism effectively mitigates a significant portion of the interference induced in the system for different period values, as shown in Figure 5.3. Summarising, the *safe_mem* and cache coloring mechanisms work well together, significantly reducing interference in the LLC and DRAM controller, which leads to an overall improvement in system performance by about 20%.

Furthermore, the relative performance for different setups is compared in Figure 5.3. Since the difference between the tested periods was minimal, a period of 1 ms and a budget of 10000 memory accesses per CPU were used. Finally, BUS ACCESS, which captures any flow of data to or from the SCU is the PMU event that generates the most effective results.

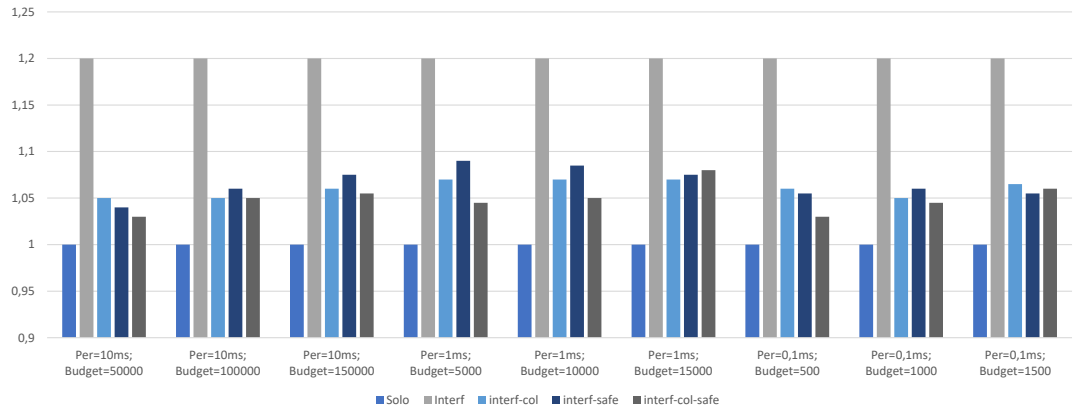


Figure 5.3: Relative Performance Overhead: Interf vs. safe_mem Algorithm (budget=100 vs. budget=10000 vs. budget=50000 vs. budget=20000).

5.1.2 MPAM Evaluation

The evaluation of the MPAM is carried out using an Arm FVP model called *RevC-AEMv8a*. This model was chosen because it implements the Armv8-A architecture and supports the MPAM processor extension, since there are no silicon platforms that implement the MPAM extension. However, the model documentation does not provide clear information about the memory hierarchy, including the specifications for the hierarchy or the sizes of the various cache levels. The evaluation is based on the conceptual design of the model shown in Figure 5.4, which includes a 32kB L1 cache for data and instructions that is private to each CPU, a 512kB L2 cache, and a 1MB L3 cache shared by all the CPUs. In terms of cache inclusion policy, if all blocks in the higher-level cache are also present in the lower-level cache, then the lower-level cache is said to be inclusive of the higher-level cache. If the lower-level cache only contains blocks that are not present in the higher-level cache, then the lower-level cache is said to be exclusive of the higher-level cache. The model in this evaluation assumes that the lower-level cache is inclusive of the higher-level cache.

Ideally, the model's conceptual design should follow the documentation provided in the reference guide of the Arm FVP manual. However, the documentation for this platform states that "A key deviation from hardware are: (...) No implementations for processor caches and the related write buffers," so an empirical analysis of the model configuration is necessary. To this end, the *MPAMF_IDR* register was analyzed to determine the implemented MPAM features and confirm the assumptions made. It has also been verified that the model implements MPAM version 1.0 through the registers *ID_AA64PFR0_EL1* and *ID_AA64PFR1_EL1*. This implies that the aforementioned register is 32-bit, which means that it is not possible to implement control settings for specific system resources of the same type, processing, or memory within one component, as it does not define resource monitors. Additionally, the model does not implement resource monitors, the MPAM error status and control registers (*MPAMF_ESR* and *MPAMF_ECR*), or MPAM error handling.

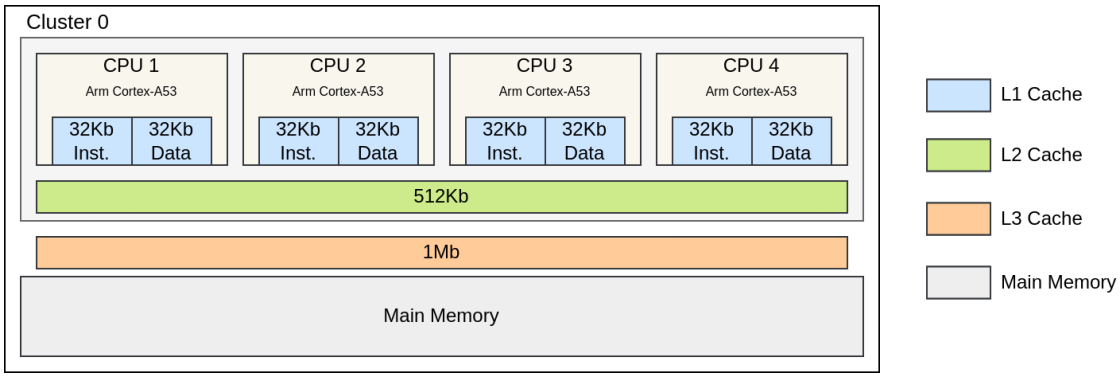


Figure 5.4: FVP Conceptual Model.

Specifically, when the value of the model parameter that specifies the size of the L3 cache (l3cache-size) is increased to 1M, it is not possible to assign it to the system VMs. It is important to note that the implementation of the L3 cache with the defined size is crucial for the MPAM implementation, as it is the LLC of the system and all MPAM configuration parameters are set at that cache level. In theory, managing/partitioning the L3 cache could help mitigate interference at the last level cache. However, it is not possible to perform a conceptual analysis using the current model to verify MPAM performance based on micro-architectural events, as was done in Section 5.1.1.

MPAM Functional Evidence

This section demonstrates the functionality of MPAM with different system configurations at the cache partitioning level, Table 5.1, which is controlled by hardware through the management of the MPAM registers responsible for controlling memory cache access by each domain of the system. The focus of this validation is to present a possible cache partitioning configuration in the system based on the capabilities of the model. To do this, cache portions are allocated to each VM, verifying the correct behavior of this processor extension in the system through its cache-related registers. Conceptually, the total capacity of the L3 cache is assumed to be 1MB, partitioned into 32 portions of 32KB each. For example, a partition configured to only allocate a maximum of four particular portions allows up to 128KB, or 1/8th of the cache's total capacity. Therefore, the evaluation setup consists of two VMs, assigning the four available CPUs . Due to model limitations, each partition is established to allocate a maximum of 50% of the overall cache capacity, which prevents each operating VM from using up half of the cache size and not sharing partitions with both VMs.

The different configurations used for this evaluation are: (i) each VM can allocate 8 cache portions, not shared (25% cache capacity); (ii) each VM can allocate 12 cache portions, not shared (33% cache capacity); (iii) each VM can allocate 16 cache portions (50% cache capacity). All cache partitioning configuration features are shown in Table 5.1.

Configuration	Cache Partition	Cache Allocated per VM
MPAM_I	VM0 - 0xF0F0	VM0 - 256kB
	VM1 - 0x0F0F	VM1 - 256kB
MPAM_II	VM0 - 0xF0F0F0	VM0 - 384kB
	VM1 - 0x0F0F0F	VM1 - 384kB
MPAM_III	VM0 - 0xF0F0F0F0	VM0 - 512kB
	VM1 - 0x0F0F0F0F	VM1 - 512kB

Table 5.1: Cache Partitioning Configurations.

Figure 5.5 also provides a visual representation that may help interpret the results. It shows the various setups and illustrates how the L3 cache is partitioned through the MPAM extension, with the understanding that it is divided into 32 portions that can be shared or not among the VMs running in the system.

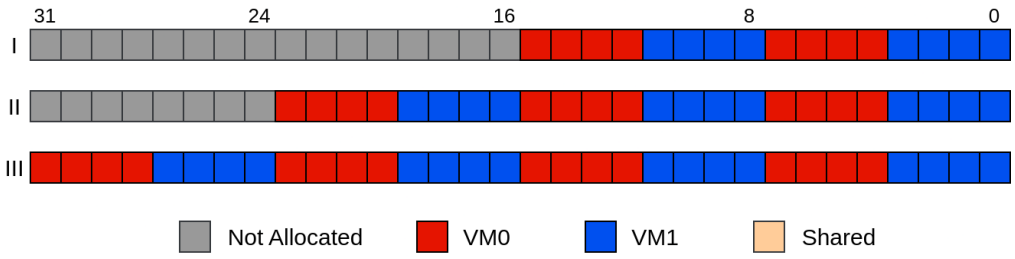


Figure 5.5: Model's L3 Cache Partitioning Based on Different Configurations.

More specifically, these results related to the assignment of cache portions to each system partition (VM) show that MPAM successfully assigns the 32-bit register *MPAMCFG_CPBM*, which provides access to the cache portion bitmap. This register is responsible for 32 portion allocation control bits that grant permission to the selected system partition to allocate cache lines within the enabled cache portions. Overall, the results of these different configurations, whether or not the cache portions are shared by the domains of the system (in this case, two VMs), demonstrate the effectiveness of the cache portion assignment, as shown in Table 5.1 and illustrated in Figure 5.5. This effectively grants isolation between the different VMs running in the system.

Test and results

The system setup used to verify the effectiveness of MPAM consists of an interference application with three cores allocated, which performs several memory accesses in order to create contention for accessing the LLC by these CPUs. The execution flow of the system leads to a performance degradation because, before accessing the main memory, the system must first access each level of the cache in sequential order, resulting in a miss at each level. Therefore, access to the main memory is slower than access to the L3 cache, which is slower than access to the L2 cache, and so on. This contention occurs due to the shared use of system resources by VMs, particularly the L3 cache and the system bus, leading to system's

unpredictability. The system setup also includes an application responsible for sampling the system PMU values to analyze and compare with the different tests performed.

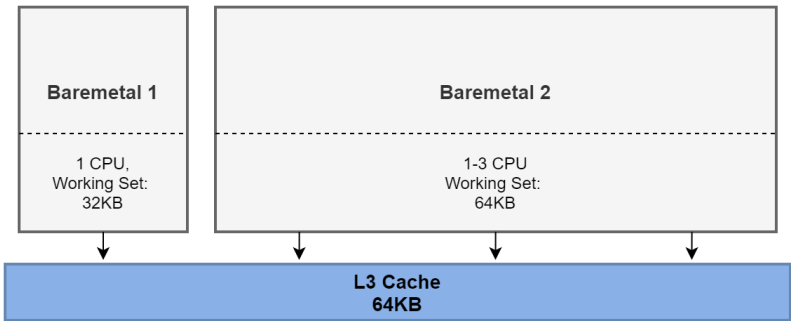


Figure 5.6: Design of the System for Testing MPAM.

Furthermore, the evaluation methodology consists of assessing the performance of the system by analyzing three different scenarios: (I) solo execution, where the baremetal runs in an isolated environment, (II) interf execution, where hosted execution is under contention, and (III) interf-MPAM execution, which adds cache partitioning capacity through MPAM to the system under contention. These different scenarios evaluate the extent to which MPAM partitioning theoretically impacts the target partitions and helps to mitigate interference. As shown in Figure 5.6, this evaluation executes two baremetal applications in separate VMs, one for benchmarking and monitoring system performance and another to induce interference by continuously writing and reading an array with a stride equal to the cache line size (64 bytes). When MPAM is enabled, the LLC (L3 cache) is partitioned according to the different configurations listed in Table 5.1. The evaluation VM working set value (baremetal 1) was selected to induce congestion in DRAM accesses, which is the aim of MPAM, avoiding congestion on the memory bus.

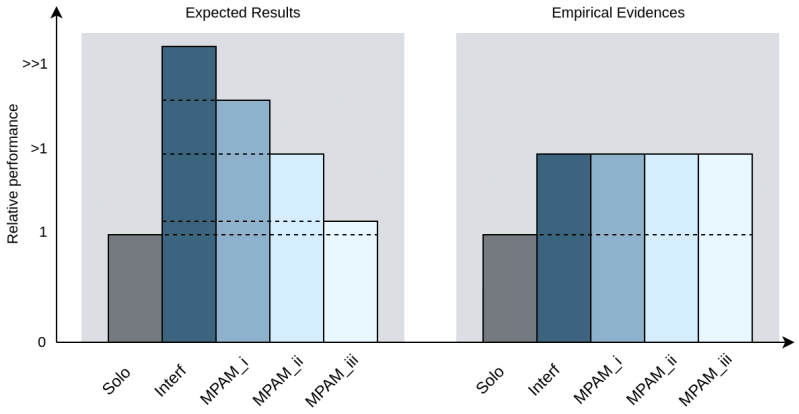


Figure 5.7: MPAM Evaluation Results.

Conceptually, the maximum configurable value for the L3 cache in this model should be 1M, but is 64kB, as increasing the "l3cache-size" model parameter to 1M is not possible for the system to assign VMs. Thus, the baremetal 1 working set is 32kB, which is 0.5x the size of the real LLC (64kB), potentially

indicating the effect of interference in the cache rather than the bus, as a large working set can result in a large number of memory accesses to the system bus.

Figure 5.7 compares the relative performance of different setups. However, the results may not align with theoretical expectations since the model does not accurately emulate the cache hierarchy. Additionally, the access to L1 and main memory have the same latency, and the memory hierarchy configuration is inconsistent. As a result, it is not possible to verify L3 cache events. Therefore, the values of the microarchitectural events analyzed are not conclusive. So, the actual values obtained do not match the expected theoretical values. The results produced by the "interf" setup and the three distinct configurations of the "MPAM_I", "MPAM_II", and "MPAM_III" setup were only equal due to the working set size selected for the monitoring VM. Therefore, no conclusions can be drawn from these results. It is worth noting that the PMU used in this model is based on the Armv8-A architecture and the model has an L1 cache size of 32 kB and an L2 cache size of 512 kB, while the L3 cache size is only 64 kB, which does not follow the typical cache hierarchy where the lowest level has the smallest size.

6. Conclusion

This section presents the conclusion of this work based on an analysis of the findings presented in the previous section. Furthermore, it provides recommendations for improving and enhancing the mechanisms developed and is divided into two main sections: (i) discussion and (ii) future work.

6.1 Discussion

This section highlights some important points about the information presented in this dissertation: (1) memory bandwidth reservation can reduce interference between domains caused by memory contention, which is implicitly limited by the memory budgeting; (2) cache coloring allows reserving a portion of the cache memory for each domain, thereby preventing mutual cache evictions; (3) since the Arm MPAM extension is relatively new and there is no physical support, this dissertation presents a comparison between this processor extension with the implementation of the above-mentioned software techniques.

Firstly, this dissertation discusses the design and implementation of an isolation mechanism for the LLC and DRAM memory controller of an Arm platform, with a particular focus on their integration within the Bao hypervisor. The "safe_mem" mechanism was integrated, taking into account that the PMU provides hardware counts of the memory accesses performed by each core for the integration of memory bandwidth isolation. A configured handler manages a periodic budget recharging event triggered by a timer and controls CPUs in budget overflow. The results of experiments using the Mibench benchmark suite evaluated the effectiveness and performance of the implemented "safe_mem" mechanism. Thus verifying a significant improvement in the hypervisor's isolation capabilities, increasing execution predictability without requiring any knowledge of the software running on other domains, and improving system performance by about 20%.

Secondly, this dissertation discusses the implementation of the MPAM API on the Bao hypervisor. The MPAM API isolates specific portions of the cache for each domain in the system, similar to how cache coloring techniques assign specific regions of the cache to particular colors. The experimental verification of the MPAM processor extension demonstrates its correct functioning, although it was not possible to fully evaluate cache contention due to limitations of the FVP model.

Overall, the developed mechanism and the MPAM API offer a solution to the contention in shared resources through the use of the "safe_mem" mechanism, which shows significant improvements in mitigating its contention.

6.2 Future Work

From the analysis of the experimental results and the conclusions in the previous section, it is possible to identify areas for improvement or extension of the developed mechanism. Although the "safe_mem" mechanism has verified, approximately, a 20% increase of system performance, mitigating the interference at memory cache level, it has some limitations, such as the static assignment of its parameters (period and budget). Therefore, future development of the "safe_mem" mechanism should consider dynamically assigning a CPU budget to each setup.

On the other hand, the MPAM API cannot be fully verified due to constraints of the FVP model and currently the existing lack of silicon and real platforms. In addition, while the API implements cache partitioning, its effectiveness in mitigating memory access contention has not been verified. In the future, it will be necessary to test its implementation directly on hardware and consider adding bandwidth management functionality such as priority partitioning, which dynamically allocates memory bandwidth to specific system domains.

Bibliography

[1] Paolo Burgio, Marko Bertogna, Nicola Capodieci, Roberto Cavicchioli, Michal Sojka, Přemysl Houdek, Andrea Marongiu, Paolo Gai, Claudio Scordino, and Bruno Morelli. A software stack for next-generation automotive systems on many-core heterogeneous platforms. *Microprocessors and Microsystems*, 52:299–311, 2017.

[2] Paolo Modica, Alessandro Biondi, Giorgio Buttazzo, and Anup Patel. Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms. pages 1651–1657, 02 2018.

[3] Marisol García-Valls, Tommaso Cucinotta, and Chenyang Lu. Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture*, 60:726–740, 2014.

[4] 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 (NG-RES)*, 2020.

[5] Heechul Yun. *Operating system level resource management for real-time systems*. University of Illinois at Urbana-Champaign, 2013.

[6] Lavanya Subramanian. Providing high and controllable performance in multicore systems through shared resource management. *arXiv preprint arXiv:1508.03087*, 2015.

[7] Sandro Pinto, Daniel Oliveira, Jorge Pereira, Nuno Cardoso, Mongkol Ekpanyapong, Jorge Cabral, and Adriano Tavares. Towards a lightweight embedded virtualization architecture exploiting arm trustzone. In *IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–4, 2014.

[8] Gernot Heiser. Virtualization for embedded systems. *Open Kernel Labs Technology White Paper*, 2007.

[9] Steven H VanderLeest and Samuel R Thompson. Measuring the impact of interference channels on multicore avionics. In *AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*, pages 1–8, 2020.

[10] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA-16 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010.

[11] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory

scheduling: Exploiting differences in memory access behavior. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 65–76, 2010.

[12] Gernot Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, pages 11–16, 2008.

[13] Jorrit N Herder, Herbert Bos, and Andrew S Tanenbaum. A lightweight method for building reliable operating systems despite unreliable device drivers. *Technical Report IR-CS-018*, 2006.

[14] Erwin Schoitsch. Design for safety and security of complex embedded systems: A unified approach. In *Cyberspace Security and Defense: Research Issues*, pages 161–174. Springer, 2005.

[15] David Cerdeira, José Martins, Nuno Santos, and Sandro Pinto. Rezone: Disarming trustzone with tee privilege reduction. *arXiv preprint arXiv:2203.01025*, 2022.

[16] José Martins, João Alves, Jorge Cabral, Adriano Tavares, and Sandro Pinto. μ rtzvisor: a secure and safe real-time hypervisor. *Electronics*, 6(4):93, 2017.

[17] Scott A Brandt, Scott Banachowski, Caixue Lin, and Timothy Bisson. Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes. In *24th IEEE Real-Time Systems Symposium (RTSS)*, pages 396–407, 2003.

[18] Phillip A Laplante et al. *Real-time systems design and analysis*. Wiley New York, 2004.

[19] Sandro Pinto, Pedro Machado, Daniel Oliveira, David Cerdeira, and Tiago Gomes. Self-secured devices: High performance and secure i/o access in trustzone-based systems. *Journal of Systems Architecture*, 119:102238, 2021.

[20] Michael Bechtel and Heechul Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 357–367, 2019.

[21] Falk Rehm, Jörg Seitter, Jan-Peter Larsson, Selma Saidi, Giovanni Stea, Raffaele Zippo, Dirk Ziegenbein, Matteo Andreozzi, and Arne Hamann. The road towards predictable automotive high-performance platforms. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1915–1924, 2021.

[22] TR Vinay and Ajeet A Chikkamannur. A methodology for migration of software from single-core to multi-core machine. In *International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*, pages 367–369, 2016.

[23] E Qaralleh, Diogo Lima, Tiago Gomes, Adriano Tavares, and Sandro Pinto. Hcm-freertos: hardware-centric freertos for arm multicore. In *IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, pages 1–4, 2015.

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

[25] PAOLO MODICA. Temporal and spatial isolation in hypervisors for multicore real-time systems. 2017.

[26] Susanta Nanda Tzi-cker Chiueh and Stony Brook. A survey on virtualization technologies. *Rpe Report*, 142, 2005.

[27] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. In *5th IEEE Consumer Communications and Networking Conference*, pages 257–261, 2008.

[28] Christoffer Dall and Jason Nieh. Kvm/arm: the design and implementation of the linux arm hypervisor. *Acm Sigplan Notices*, 49(4):333–348, 2014.

[29] Daniel Oliveira, Tiago Gomes, and Sandro Pinto. utango: an open-source tee for iot devices. *IEEE Access*, 10:23913–23930, 2022.

[30] Nicolas Dagieu, Alexander Spyridakis, and Daniel Raho. Memguard: A memory bandwith management in mixed criticality virtualized systems memguard kvm scheduling. In *10th Int. Conf. on Mobile Ubiquitous Comput., Syst., Services and Technologies (UBICOMM)*, 2016.

[31] Zahir Hussain Shah. *Windows Server 2012 Hyper-V: Deploying the Hyper-V Enterprise Server Virtualization Platform*. Packt Publishing Ltd, 2013.

[32] Ankita Desai, Rachana Oza, Pratik Sharma, and Bhautik Patel. Hypervisor: A survey on concepts and taxonomy. *International Journal of Innovative Technology and Exploring Engineering*, 2(3):222–225, 2013.

[33] Samuel Pereira, Joao Sousa, Sandro Pinto, José Martins, and David Cerdeira. Bao-enclave: Virtualization-based enclaves for arm. *arXiv preprint arXiv:2209.05572*, 2022.

[34] Ayoosh Bansal, Rohan Tabish, Giovani Gracioli, Renato Mancuso, Rodolfo Pellizzoni, and Marco Caccamo. Evaluating the memory subsystem of a configurable heterogeneous mpsoc. In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, volume 7, page 55, 2018.

[35] Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Computing Surveys (CSUR)*, 48(2):1–36, 2015.

[36] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54, 2013.

[37] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *24th Euromicro Conference on Real-Time Systems*, pages 299–308, 2012.

[38] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[39] Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. Look mum, no vm exits!(almost). *arXiv preprint arXiv:1705.06932*, 2017.

[40] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, 2014.

[41] Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, 2014.

[42] Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. E-warp: A system-wide framework for memory bandwidth profiling and management. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 345–357, 2020.

[43] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. Coloris: a dynamic cache partitioning system using page coloring. In *23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 381–392, 2014.

[44] Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. A comparative study of predictable dram controllers. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(2):1–23, 2018.

[45] Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Drambulism: Balancing performance and predictability through dynamic pipelining. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 82–94, 2020.

[46] Matteo Zini, Daniel Casini, and Alessandro Biondi. Analyzing arm's mpam from the perspective of time predictability. *IEEE Transactions on Computers*, (01):1–14, 2022.

[47] Arm Ltd. ARM Cortex -A53 MPCore Processor Technical Reference Manual. Accessed on: December 15, 2021. [Online] Available: https://developer.arm.com/documentation/ddi0500/latest/.

[48] Arm Ltd. Fixed Virtual Platforms (FVP) Reference Guide. Accessed on: December 2, 2021. [Online] Available: https://developer.arm.com/documentation/100966/1118/.

[49] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pages 3–14, 2001.

[50] Thivanon Kansuwan and Thawatchai Chomsiri. Authentication model using the bundled captcha otp instead of traditional password. In *2019 joint international conference on digital arts, media and*

technology with ECTI northern section conference on electrical, electronics, computer and telecommunications engineering (ECTI DAMT-NCON), pages 5–8, 2019.

[51] Arm Ltd. AArch64 Programmer's Guides Generic Timer. Accessed on: October 30, 2021. [Online] Available: https://developer.arm.com/documentation/102379/0000/?lang=en.

[52] Arm Ltd. Arm Architecture Reference Manual Supplement - Memory System Resource Partitioning and Monitoring (MPAM), for Armv8-A. Accessed on: March 20, 2022. [Online] Available: https://developer.arm.com/documentation/ddi0598/latest.