Universidade do Minho
Escola de Engenharia

Pedro Gonçalo Oliveira Leite

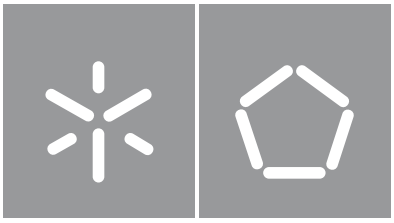
Contention and predictability study on Arm Cortex-M TrustZone-enabled MCUs

Maio de 2021

Contention and predictability study on Arm Cortex-M TrustZone-enabled MCUs

Pedro Leite

UMinho | 2021



Universidade do Minho
Escola de Engenharia

Pedro Gonçalo Oliveira Leite

Contention and Predictability study on Arm Cortex-M TrustZone-enabled MCUs

Dissertação de Mestrado
Mestrado Integrado em Engenharia Eletrónica Industrial e Computadores
Sistemas Embebidos e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Sandro Emanuel Salgado Pinto

Maio de 2021

Agradecimentos

Aos meus pais e padrastro, quero dirigir as minhas primeiras palavras de agradecimento. Todo o vosso suporte, amor e confiança incondicional foram indispensáveis no meu percurso académico. Vocês conseguiram sempre reunir e proporcionar-me todas as condições necessárias para o meu desenvolvimento enquanto pessoa. Obrigado, por todas as oportunidades que tornaram esta dissertação realidade.

Aos meus dois irmãos, Tiago e a pequena Maria, que são tudo para mim, queria agradecer por todo o seu apoio e amor que me deram ao em toda esta jornada. Obrigado pequena por conseguires sempre animar-me e dar-me um grande sorriso, mesmo nos tempos mais sombrios.

Aos meus avós, que cuidaram e me viram crescer, quero também agradecer do fundo do meu coração. Obrigado por toda a paciência, amor e dedicação ao longo destas duas décadas e meia.

Gostaria de agradecer ao meu orientador e professor pelo apoio e suporte contínuo, doutor Sandro Pinto, no desenvolvimento deste longo e árduo trabalho. Todo o seu conhecimento que me foi transmitido nesta dissertação demonstraram-se ser fulcrais para o seu término.

Queria também deixar uma especial palavra de agradecimento ao mestre e futuro doutor, Daniel Oliveira, por todo o seu rigor e conhecimento que transmitiu, não exigindo nada menos do que o melhor de mim. Agradeço também pela sua motivação e pensamento crítico, que me guiaram e ajudaram em tempos de maior dificuldade. Obrigado por me teres ajudado a crescer como pessoa e profissional.

Por último, queria deixar um especial agradecimento aos meus amigos, Daniel, Luís, Tiago, Ricardo, Sérgio e Zé Rui, que foram uma parte integrante desta minha caminhada, tendo ela os seus ”altos” e ”baixos”. Um especial agradecimento também a uma truta especial, Edgar, pela sua enorme dedicação na procura de uma figura de extrema importância.

A todos que suportaram de maneira direta ou indireta o desenvolvimento desta dissertação, um grande obrigado!

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

Com a evolução tecnológica e a sua produção em massa, os sistemas embebidos tornaram-se uma necessidade no dia-a-dia da sociedade. Com a redução do seu custo de produção e, continua inovação a nível de semicondutores, estes sistemas tornaram-se mais acessíveis, baratos e adaptáveis a necessidades especificas.

Em anos passados, os sistemas embebidos consistiam em pequenos dispositivos conectados responsáveis por executar uma tarefa necessária para a operação de um sistema de maior dimensão. Além disso, estes sistemas de maior dimensão, eram baseados em arquiteturas federadas, que consistiam em separar os vários subsistemas em diferentes unidades de processamento. No entanto, devido ao rápido aumento dos requisitos e número de funções, as arquiteturas federadas começaram a tornar-se impraticáveis devido ao tamanho, peso e consumo energético, criando a necessidade de integrar múltiplos *worloads* com diferentes criticidades em uma só plataforma.

De modo a assegurar isolamento espacial e temporal, a comunidade científica propôs, com recurso as diferentes extensões de virtualização e segurança, soluções como hipervisores e ambientes de execução segura.

No entanto, à medida que arquiteturas *multi-core* ganham força no mundo embebido, é iminente o surgimento de novas fontes de imprevisibilidade, devido ao seu acoplamento a hierarquias de memória partilhadas, levando inerentemente ao surgimento de contenção quando recursos são partilhados (e.g., *caches*, bancos de memória).

Diferentes classes de soluções que abordam várias fontes de imprevisibilidade têm sido propostas pela comunidade científica e, que apesar de serem altamente eficazes para sistemas *high-end*, plataformas *low-end* não apresentam mecanismos de capazes de recolher e manipular informações tão detalhadas sem o uso de ferramentas externas.

Palavras chave: execução determinística, previsibilidade, recursos partilhados, contenção, arquiteturas *multi-core*.

Abstract

Due to the evolution of technology and mass production, embedded systems are ubiquitous in modern infrastructures and society. With manufacturing costs decreasing and the continous innovations in semi-conductor technology, cloud computing, mobile connectivity, and high computing capabilities, these systems have become more accessible, affordable, and flexible.

In the early years, embedded systems were small connected devices responsible for executing an individual task of a more powerful system. Moreover, these bigger systems were based on federated architectures, which physically separated several subsystems across different computing units. However, with the rapid increase of requirements and the number of functionalities, federated architectures became impracticable due to their size, weight, and power consumption (SWaP-C), forcing a shift to the implementation of multiple workloads in the same system.

To assure spatial and temporal isolation, the academia community has proposed different solutions such as hypervisors and Trusted Execution Environments (TEE), supported by virtualization or security extensions.

However, as multi-core architectures pave the way into the embedded sector to deliver more computing power, new sources of indeterminism are likely to appear. Such platforms are based on multiple cores that are tightly coupled with a shared memory hierarchy, which inherently leads to contention problems over these shared hardware resources. Therefore, the additional sources of performance unpredictability are typically related to shared resources, which include elements from the memory hierarchy (e.g., shared caches, memory controllers, memory banks, and I/O devices) to the system bus.

The scientific community proposed different classes of solutions that tackle different sources of unpredictability. But, despite the proposed solutions being highly effective, low-end platforms do not present hardware mechanisms capable of collecting such meticulous information without external tools.

Keywords: deterministic execution, predictability, shared resources, contention, multi-core architectures.

Contents

List of Figures

List of Tables

List of Abbreviations and Acronyms

AMP Asymmetric Multi-Processing

DWT Data Watchpoint and Trace unit

eSRAM External Static Random Access Memory

ETB Embedded Trace Buffer

ETM Embedded Trace Macrocell

iSRAM Internal Static Random Access Memory

IDAU Implementation Defined Attribuition Unit

IRQ Interrupt Request

ITM Instruction Trace Macrocell

I/O Input/Output

LLC Last Level Cache

MCU Microcontroller

MHz Mega-Hertz

MHU Message Handling Unit

MMU Memory Managment Unit

MPC Memory Protection Controller

MPU Memory Protection Unit

NS Non-Secure

NSW Non-Secure World

NSVW Non-Secure Virtual World

NVIC Nested Vector Interrupt Control

PMC Performance Measuring Counters

P.D. Performance Degradation

R.P.D. Relative Performance Degradation

SAU Security Attribuition Unit

SCR System Control Register

SoC System-on-Chip

SWaP-C Size, Weight and Power Consumption

SW Secure World

TCB Task Control Block

TCM Tightly Coupled Memory

TEE Trusted Execution Environment

TPIU Trace Port Interface Unit

VM Virtual Machine

VTOR Vector Table Offset Register

VCPU Virtual CPU

WCET Worst-Case Execution Time

WCB World Control Block

WFE Wait for event

WFI Wait for interrupt

Chapter 1

Introduction

Due to the evolution of technology and mass production, embedded systems are ubiquitous in modern infrastructures and society. With manufacturing costs decreasing and the continous innovations in semi-conductor technology, cloud computing, mobile connectivity, and high computing capabilities, these systems have become more accessible, affordable, and flexible [13]. According to Arm, it is expected that between 2017 and 2036, a trillion new internet of things (IoT) systems will be produced [14]. This exponential growth will shake the industry's standards and trigger a revolution in modern software design.

In the early years, embedded systems were small connected devices responsible for executing an individual task of a more powerful system. Moreover, these bigger systems were based on federated architectures, which physically separated several subsystems across different computing units. For years, this architecture has governed these systems' design with significant benefits, most notably the inherent isolation between workloads [15]. However, with the rapid increase of requirements and the number of functions, federated architectures became impracticable due to their size, weight, power consumption and cost (SWaP-C) [16]. Therefore, to diminish the overall system SWaP-C and to take full advantages of multi-core chips, new designs were proposed where multiple applications with different criticalities requirements (e.g., real-time, performance, security, safety) are integrated into a single platform, i.e., mixed-criticality systems [17, 18].

In this scenario, industries such as automotive and aerospatial adopted virtualization technology to manage the complexity of mixed-critical workloads (e.g., real-time, safety, security requirements) and reduce costs (i.e., SWaP-C requirements). From the application point of view, when virtualized, it is given the impression of total control of the system resources while the unknowing of an existent underlying a hypervisor that partitions the available resources between the multiple machines, and ensures their spatial and temporal isolation [19, 20, 21]. However, most commerical off-the-shelf (COTS) IoT-enabled devices lack the hardware features to provide the necessary isolation mechanism exploited by virtualization to ensure

separation between the consolidated workloads. Furthermore, most of these microcontroller devices are based on Arm architectures [22, 23], which, until recently, did not provide the hardware infrastructures to develop virtualization solutions targetting these systems. However, with the release of the ARMv8-M architecture [4, 1], Arm introduced the Arm TrustZone technology [24] in their low-end systems portfolio. Solutions based in TrustZone appear targetting different reliable infrastructures to IoT-based devices, such as hypervisors [16, 25, 26, 27] and Trusted Execution Environments (TEEs) [28, 29, 30], focused on creating environments where temporal and local isolation is key, reducing/preventing data leaking coming from multiple types of malicious attacks to sensitive information [31].

With the continuous pushing for systems with better performance, multi-core architectures are rising, even in the low-end embedded sector. However, such architectures create new challenges and difficulties derived from the reciprocal interference caused by the sharing of resources (e.g., memory controllers, system buses, cache) [32], which reduces the determinism and, therefore, the predictability of the system. Moreover, when these systems tend to consolidate mixed-critical requirements, such issues become more relevant because of hard real-time requirements that must be met. Therefore, over the years, the real-time community has been proposing different classes of solutions to minimize contention and increase predictability (e.g., cache coloring [33, 7], memory bandwidth reservation [9, 8], redesign memory controllers [34, 35]). However, as of this writing, these proposed solutions depend on the existence of hardware features (e.g., two-stage MMU and performance counters) that are not available in resource-constrained devices (e.g., the low-end Cortex-M family of Arm's microcontrollers). Hence, this master thesis aims at developing a predictable resource-sharing mechanism targetting low-end microcontrollers.

An in-depth study was performed to assess which hardware components featured in Arm-based architectures were suitable candidates to support the proposed mechanism. From that assessment, it was identified debug and tracing macrocells, capable of capturing the resource usage by the cores, that were later used to develop a mechanism, called Keep It Real (KIR), focused on tackling the contention problems faced in modern platforms. The solution was integrated and further evaluated under an in-house TEE system, which neglects resource sharing contention and lacks temporal isolation between the executing environments. The results show that the developed mechanism improves the evaluated systems' determinism and predictability significantly.

1.1 Problem Statement

IC Insights forecasts that in the next few years, the microcontroller market will reach a new next record-high level of revenues [36]. For these latest microcontrollers, performance is vital and obtained at the expense of predictability. More raw computing power enables the opportunity to integrate multiple applications into the same platform by exploring all hardware features that the device has to offer. However, the consolidation of different workloads into one single platform results in systems with many applications that vary from soft to hard real-time requirements [37, 38].

Typically, to ensure the predictability of a system, timing constraints are enforced on system tasks after proper tests are performed on worst-case execution time (WCET) estimations. However, in real scenarios, precise estimation of WCETs is extremely difficult due to several low-level architectural mechanisms (e.g., interrupts, direct memory accesses (DMA), pipelines, caches, hardware prefetching, out-of-order execution), which introduce non-deterministic behavior, making it impracticable to precisely estimate the system's timing constraints [39, 40]. Moreover, as multi-core architectures pave the way into the embedded sector to deliver more computing power, new sources of indeterminism are likely to appear. Such platforms are based on multiple cores that are tightly coupled with a shared memory hierarchy, which inherently leads to contention problems over these shared hardware resources. Therefore, the additional sources of performance unpredictability are typically related to shared resources, which include elements ranging from the memory hierarchy (e.g., shared caches, memory controllers, memory banks, and I/O devices) to the system bus. In fact, an inter-core dependency phenomenon occurs as the memory accesses from one core can be influenced by the requests from other cores [9, 41, 42, 43]. To further complicate the problem, memory controllers are either not flexible enough to handle Systems on Chip (SoC) increasing complexity or do not support analytical design-time verification of hard real-time requirements.

Memory controllers with static scheduling may increase the predictability of the system, but at the same time, reduces the flexibility to changes in the number of requests, making it harder to distinguish latency requirements of critical requests without memory over-allocating [35, 34]. Moreover, conventional multi-core architectures tend to implement a shared bus to connect the cores and the shared memory, meaning that communication between the cores or accesses to memory is done through the a single channel. Therefore, the bus is locked once a single access is done, leading to serious contention [41, 44].

At the last level of cache, when shared, it is introduced unknown cache states since one core can cause the eviction of cache lines used by another core to accommodate the data of its domain, resulting

in mutual increase memory access times [8, 7]. Also, multiple workloads performing input/output (I/O) operations, with high variability, leads to I/O bursts, causing network and file systems contention, and therefore jeopardizing the performance and predictability of the system [45].

The scientific community proposed different classes of solutions that tackle different sources of unpredictability on these systems: (i) at the cache level, cache coloring is a technique used to partition this memory across different workloads [38, 8, 20, 7]; (ii) at the memory controller, several scheduling policies were proposed [35, 34, 10]; (iii) at the main memory level, by using techniques to regulate the access bandwidth of multiple cores [9, 8, 33]. Despite the proposed solutions being highly effective, low-end platforms do not present hardware mechanisms capable of collecting such meticulous information without external tools. Therefore, at the time of writing, mechanisms capable of ensuring predictability and determinism in constrained devices are none existent, meaning this is still a grey area of the field that requires proper attention. In accordance with the preliminary study presented in Section 3.2, it is verified that low-end platforms also suffer from the same type of problems presented previously.

1.2 Aims and Scope

This master dissertation's primary goals is an in-depth study of contention points caused by shared resources in low-end platforms that can jeopardize the overall system's predictability. After identifying major contention points on the target platform, the study will focus on target-specific hardware modules to support a mechanism designed to improve the system's predictability. Therefore, guided by the conducted study and the knowledge gathered during the literature review, this dissertation's primary output is to design, develop, and evaluate in a real-scenario such mechanism. Since the low-end embedded sections is vast, the focus was narrowed to Armv8-M-based architectures, which cover the most recent platforms of the microcontroller market. Regarding the evaluation, KIR will be integrated and deployed in an in-house TEE (i.e., uTango). uTango [6] is a multi-world TEE for Armv8-m-enabled (with TrustZone extensions [4, 1]) IoT devices which enables the execution of multiple environments within strongly isolated compartments. As of this writing, uTango supports only single-core execution; thereby, uTango needed to be extended to multi-core execution and evaluated afterwards the system integrated. It is expected that with KIR, uTango achieves better determinism and improves significantly the system predictability.

The set of goals described during this section are outlined below:

1. in-depth study of interference mitigation techniques caused by sharing system resources and their applicability on low-end platforms;
2. evaluation of possible contention points and their impact on the predictability of the system;
3. design and development of a mechanism ,i.e., KIR, able to actuate in contention created by shared resources;
4. experimental evaluation of KIR and its improvements of the system predictability and determinism;
5. porting uTango from single-core to a multi-core scenario;
6. to evaluate the KIR mechanism under the uTango TEE.

1.3 Dissertation's Structure

The present dissertation starts with a contextualization (Chapter 1) about the new trends and challenges that the industry faces today and the collaterally generated problems attached to the search of higher performant systems, leading to increased jitter in shared resources under various workloads. Towards the end of this chapter, the proposed goals are explained.

Chapter 2 encloses two main sections, which present: (i) background knowledge needed for the development of this dissertation; (ii) related work developed by the scientific community that addresses similar problems to the one target in this dissertation. In the first phase, concepts regarding the Arm architecture used are described. This section also mentions the recent introduction of the TrustZone technology to Cortex-M Microcontrollers and the changes introduced to the architecture. Moreover, there is a focus on the significant potentialities and drawbacks given by Debug and Tracing (i.e., CoreSight architecture) features spanned across Armv8-M-based microcontrollers. In the end of the chapter, it is explained solutions derived from the virtualization and security extensions that target mixed-criticality systems. With the combination of the computer architectures, memory layouts, predictability issues derived from increased performance, and the need for mixed-criticality systems, a clarified picture of the big problem is presented. Regarding the second phase, it gathers several solutions proposed by the scientific community that tackle contention and predictability issues but targetting different platforms (i.e., high-end devices).

Chapter 3 explains the overall system specification. It starts with a more in-depth platform description and tools used for the development and evaluation of the proposed solution. Moreover, it is presented

an illustrative study with factual results of the presented platform used in the dissertation. After, it is described the mechanism architecture and the modules needed to build it. Therefore, it is specified in detail the operation and functionality of each module composing the mechanism. After, it is explained the integration of the solution on the in-house uTango TEE, and decisions taken to transition from a single-core to a multi-core system. In the last part of the chapter, it is laid down the set of benchmarks proposed to test each one of the mechanism functionalities and its integration with uTango.

Chapter 4 focuses on the implementation details regarding the development of the proposed KIR mechanism. Lastly, it is described how uTango was extended from a single-core to multi-core architectures, and how the mechanism was integrated.

In Chapter 5 it is presented the results regarding the proposed benchmarks, focused on the evaluation of the mechanism capability of improving the system's predictability. Lastly, it is assessed the mechanism functionalities after integration with uTango, and compared with previously obtained results.

Lastly, Chapter 6 closes this dissertation by resuming the developed work and concludes by discussing the achieved results and limitations. Moreover, some suggestions aim to improve the developed work are presented.

Chapter 2

State of the Art

This dissertation proposes investigating the main predictability bottlenecks originated by contention points during access of hardware shared resources on low-end Arm-based devices. Such a research topic has been observed in the last years on high-end platforms by the academic community; thereby, Section 2.2 will cover the most prominent works that propose different solutions to tackle the issue. That subsection is divided in three main categories, according to the taxonomy of the proposed solution. Before reviewing those academic works, the present chapter exposes in Section 2.1 fundamental concepts essential to design and develop the KIR mechanism. This mechanism aims to restore the determinism and improve Arm-based systems' predictability (more specifically Armv8-M). Therefore, topics related to the Armv8-M architecture (e.g., programming model, TrustZone security extensions, debug and tracing) will be covered.

2.1 Background

This section will address some fundamental concepts surrounding several technologies needed for the development of this dissertation. First, it is introduced the underlying building blocks of Armv8-M architecture in Section 2.1.1. Throughout the section it is explained the architecture programming model, exceptions and interrupts, and the power management key registers used. Furthermore, it is detailed the main components used to partition resources under TrustZone-enabled microcontrollers. Moreover, still within the Armv8-M topic, it is explored the key components used to debug and trace Arm Cortex-M processors, ranging from non-invasive to extremely invasive techniques. Lastly, there is a dive into the TrustZone technology and the solutions that arouse around them, i.e., TEEs such as uTango; thereby, an overview of uTango architecture is also presented.

2.1.1 Armv8-M Architecture Overview

Over the years, the ARM Cortex-M Processors became mainstream in a wide range of embedded applications such as sensors, wireless communication application-specific integrated circuits (ASIC), power management integrated circuits (IC) and a part of more complex SoCs [4, 46]. Depending on the target application, Arm offers three different families of processors [47]: (i) Cortex-A family focuses for their use in high-end devices capable of running an operating system (OS) such as Linux and Android (e.g., smartphones, tablets); (ii) Cortex-R targets high-performance real-time applications (e.g., hard disk controllers, network equipment); (iii) Cortex-M is a family of microcontroller processors designed to have a lower silicon footprint and a high energy-efficiency.

Architectural specifications define the behavior of the processor from the software and debug point of view. Therefore different processor implementations can result from the architecture. The existing Cortex-M processors are based on two architecture versions named Armv6-M [22] (e.g., Cortex-M0, Cortex-M1), which is used for processor designs focusing ultra power designs and, Armv7-M [23] (e.g., Cortex-M3, Cortex-M4, Cortex-M7) are more used in mainstream microcontroller products and higher performant embedded systems [4, 1]. To succeed them, Arm has created the Armv8-M architecture, which remains a 32-bit architecture and with high compatibility with the last architectures (Armv6-m and Armv7-m) so that migration of software within the Cortex-M family became easier. Arm partitioned the Armv8-M architecture into two different profiles to address the same needs as the last architectures did, but with slight enhancements [1, 46]: (i) Armv8-m Baseline similar to Armv6-M; (ii) Armv8-M Mainline equivalent to Armv7-M. Enhancements came in the form of better debug capability (e.g., breakpoint and watchpoint units), support for more interrupts, newer programmer's model for the memory protection unit (MPU) and the implementation of the security extension named TrustZone Technology [46, 4, 1].

The TrustZone technology adds a new state to the processor, creating two different states, coexistent inside the core, named secure and non-secure, where secure resources are isolated from the non-secure ones. Moreover, when the security extension is implemented, secure applications can access non-secure resources (e.g., non-secure SysTick) by using the non-secure memory aliases. Moreover, to ensure that legacy code can be easily portable, the internal register memory-mapping are equal [4, 1, 48, 3]. Although TrustZone was recently implemented in the Cortex-M family, his presence was already standard in the Cortex-A processors family. Notwithstanding both TrustZone architectures (i.e.,TrustZone-A and TrustZone-M) are similar on the high-level concept, there are major differences in how the security extension is

implemented in these two different families of processors, namely the fast context-switch between the various states of the core. Hence, due to the necessity of deterministic behavior, low power consumption, and low interrupt latency, Arm redesigned the security extension for the Cortex-M family of processors [48, 47, 49].

Orthogonal to the security states, there are the execution modes (handler and thread) and privilege levels (privileged and unprivileged), as it is shown in the Figure 2.1. Thread mode refers to the execution of the application code, which can be privileged or not, while in handler mode, that is used to deal with system interrupts and exceptions, it is always at a privileged level. Being on a privileged level allows the application to have access to all of the available resources and instructions, while in unprivileged the core has limited access to the instructions that it can perform (e.g., MSR and MRS) and to resources (e.g., System Timer, Nested Vector Interrupt Control (NVIC), and System Control Block) [3, 1].

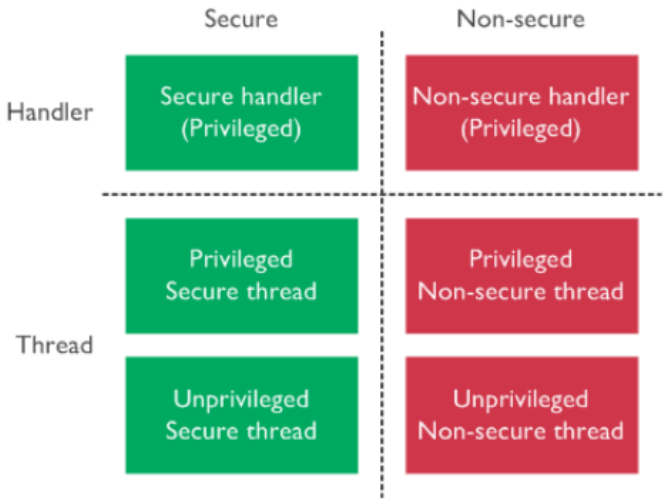


Figure 2.1: Armv8-M processor modes, states, and privileged levels [1].

2.1.1.1 Programming Model

The Arm architecture has sixteen 32-bit registers that can be used from the application point of view (R0-R15). From the available registers, thirteen of them can be used as general-purpose registers (R0-R12). From the last ones, R13 is used as the stack pointer (SP), the R14 is used as the link register (LR), so that information can be restored after a return of a function, exception, or any subroutine, and R15 as the program counter (PC) that holds the next instruction address. In the state of reset, the processor loads the PC with the value of the reset vector. The stack used decrements from the starting point, meaning that

the stack pointer holds the address of the last stacked item in memory. Each push leads to a decrement of the R13 and a write into the new memory location. Physically there are two stack pointers: (i) Main Stack Pointer (MSP); (ii) Process Stack pointer (PSP). The existence of two stack pointers allows the end-user to implement a multi-tasking operating system, where the PSP can be used for individual tasks and the MSP for the Kernel. Furthermore, when an interrupt or exception occurs, the processor, if using the PSP, pushes its value into the stack and switches to the MSP. In case the Security Extension is implemented, there are two stack points for each state of the processor state and they are banked in case of a transition, forcing a separation between the secure and non-secure stacks [1, 3].

Additionally to the architecture, there are special designed registers known as Combined Program Status Register (XPSR), Exception Mask registers, and the CONTROL register. The XPSR consists in the concatenation of the: (i) Application Program Status Register (APSR) that contains the current state of the condition flags of previous executed instructions; (ii) Interrupt Program Status Register (IPSR) that contains the exception type number of the current Interrupt Service Routine (ISR); (iii) Execution Program Register (EPSR) that contains the *Thumb* state bit.

Within the Exeception Mask registers there are (i) the Priority Mask Register (PRIMASK) Priority Mask Register (PRIMASK) prevents the servicing of all exceptions with configurable priority; (ii) the Fault Mask Register (FAULTMASK) that prevents the servicing of all exceptions except the Non-maskable Interrupts (NMI), HardFaults, or Resets; (iii) the Base Priority Mask Register (BASEPRI) used to define the minimum group priority for exception processing (i.e. if set to a nonzero value, it prevents the servicing of all exceptions with the same of lower group priority as the value of BASEPRI). All exceptions have an associated priority, where a low value means a high priority. The CONTROL register is used to control the privilege level (bit 0), the currently used stack pointer (bit 1) and, also indicates whether the Floating-point Extension is currently active [1, 3]. All of the Exception mask registers, the bits 1 and 0 of the CONTROL register, and Vector Table Offset Register (VTOR) are banked between the different states.

2.1.1.2 Resources Partition

On TrustZone-M-based systems, the memory space can be partitioned according to the attributes of two units: (i) Implementation-Defined Attribution Unit (IDAU) that provides static address partitioning and supports up to 256 non-programmable regions. (ii) Security Attribution Unit (SAU) which allows dynamic configuration of the memory addresses in multiple regions (e.g., 8 regions). The security of the memory regions (i.e., Secure, Non-secure and Non-secure Callable) are determined by a logic *OR* operation

between the IDAU and SAU. Furthermore, in TrustZone-enabled microcontrollers there are other components, called *security gates*, used to ensure the overall systems's security. The *security gates* act as firewalls on TrustZone-oblivous slaves, and are controlled by a central system security controller, specified by sillicon providers (e.g., Arm Musca-A1 Memory Protection Controllers (MPC)) [24, 4, 1]. Using the TrustZone-aware MPU, it is possible to create privileged memory zones. The MPUs are banked between states [4, 1].

2.1.1.3 Exceptions and Interrupts

Exceptions and interrupts are conditions or system events that require the core to stop its normal execution and start a specific routine, known as a handler, in a privileged mode to ensure the normal function of the system. In the Armv8-M processor families, in contrary to the other Arm families, there is an integrated NVIC, which identifies external interrupts to the core. Exceptions are identified by their (i) number that represents an offset within the Vector Table, where the name set equals to the one of the handler; (ii) priority number, where the lower the number, the higher its priority. The vector table is at an implementation-defined address (e.g., 0x00000000) and, if in a privileged level, can be relocated using the vector table offset register (VTOR) [50].

Within the Armv8-M-base architecture, there are 15 exceptions, all with different purposes (e.g., deal with reset events): (i) treat an error that occurs when processing other exceptions (e.g., NMI), or to catch a system failure (e.g., divisions by zero, access to protected memory); (ii) respond to a debug event caused by a debug or tracing unit (e.g., Debug Monitor); (iii) if a non-secure application tries to access secure resources without permission (e.g., SecureFault); (iv) when it is executed an *Supervisor Call* instruction (SVC); (v) the SysTick timer reaches zero. Extra to the core, there are the interrupt requests (IRQ) that can be triggered by an external peripheral or generated by software [50, 3].

In the case of TrustZone-enabled microcontrollers, the interrupts can be set as secure or non-secure by configuring the Interrupt Target Non-secure (ITNS) register present in the NVIC. The Arm's Cortex-M architecture supports automatic hardware stacking and un-stacking of some CPU registers to reduce the latency of exception entrance. To avoid any leakage, when dealing with events that have a different state than the current state of the CPU (e.g., CPU in the secure state receives an interrupt triggered by a non-secure peripheral), all non-banked registers are pushed into the stack and its contents erased. Since the VTOR and exception handlers are banked between states, secure and non-secure interrupts can share the

same priority level. Moreover, the security interrupts can be programmed to have a higher priority than the non-secure ones to avoid any type of attacks (e.g., denial-of-service) [4, 50, 3].

2.1.1.4 Power Management

One of the Cortex-M processor families' features is the capability to reduce power consumption by changing from active mode to one of the available sleep modes. Typically, there are two types of sleep modes, named *sleep* and *deep sleep*. During *sleep* mode, most of the clock signals are stopped, and only some parts of the processor are running, while in *deep sleep*, all clock signals are disabled [51, 47, 52]. The *sleep* mode can be selected by the bit SLEEPDEEP in the System Control Register (SCR). Moreover, the architecture also features two instructions that can enable the sleep modes, i.e., WFI (Wait For Interrupt) and WFE (Wait For Event). Both instructions suspend the execution and put the processor in the lowest power mode available until it receives a reset, or an asynchronous exception, or any other event occurs. The difference between the WFI and WFE instruction is that the WFE is conditional and WFI is unconditional, i.e., the CPU to enter a sleep mode when using the WFE, it is required some condition to be respected, more specifically, the event register's value. Furthermore, the CPU can enter a sleep mode by changing the bit SLEEPONEXIT to one. When the SLEEPONEXIT is set, the CPU first executes all of the pending exceptions and enters a *sleep* mode on the exit of the last one [51, 3].

The procedure to wake up the processor depends on the mechanism that caused the *sleep* mode. If the processor's cause to enter a *sleep* mode was a WFI instruction, or by setting the SLEEPONEXIT bit, the processor wakes up if it detects any exception with enough priority to cause exception entry and, the PRIMASK bit must is set to 1, and the FAULTMASK bit set to 0. If the cause was a WFE instruction, the processor wakes up if (i) it detects an exception with enough priority to cause an exception entry, ignoring PRIMASK; (ii) it detects an external event signal; (iii) another core of the system executes a SEV instruction which signals an event to all cores of the system. Furthermore, if the SEVONPEND bit in the SCR is set to 1 any new pending interrupt also triggers an event and causes the processor to wake up, even if the interrupt is disabled or has insufficient priority to cause exception entry [51, 3].

2.1.1.5 Debug and Tracing

Debugging is an essential part of software development. It enables software developers to collect crucial data about the program's behavior and detect the cause of any problem [53]. There is a critical difference between debug and trace since debug refers to the ability to observe or modify register values

of processors or peripherals while tracing provides a continuous collection of system information for later analysis [54]. Thus, it is possible to associate debug to an invasive type of debugging and tracing to a non-invasive debugging (Figure 2.2). Invasive debug can be performed in two methods: halting or monitoring the core. The halting debug halts the cores when some event cores (e.g., Debug Exception). This kind of invasive debug is usually done by an external tool such as a debugger connected to the platform via an external interface. Monitor debug implies that core must handle itself the exception, when raised, without the intervenience of third-party tools. Regarding tracing methods, they are done by profiling macrocells existent in architecture (e.g., Embedded Trace Macrocell), that observe the core's behavior and at the same time can export the information to an external tool [53].

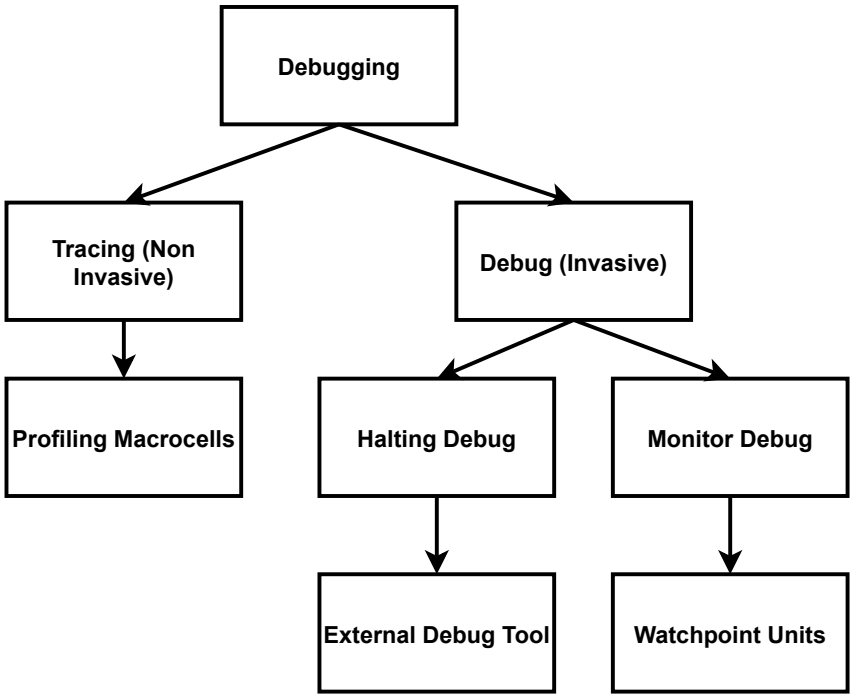


Figure 2.2: Debugging types.

The Cortex-M processors' debug architecture is based on the Arm CoreSight architecture, which scales easily and supports multi-processor systems [47]. The CoreSight is usualy divided into multiple components with different functions [55, 56] (see Figure 2.3):

- Debug and Access Port (DAP) that controls and allows configuration of the tracing units;
- Intruction Trace Macrocell (ITM) and Embedded Trace Macrocell (ETM) to generate trace data;
- CoreSight Trace Funnel (CSTF), Cross Trigger Network (CTN) or a Replicator to provide connections, triggering and control of the traced data;

- Trace Port Interface Unit (TPIU) and Embedded Trace Buffer (ETB) sinks that function as end points for trace data;

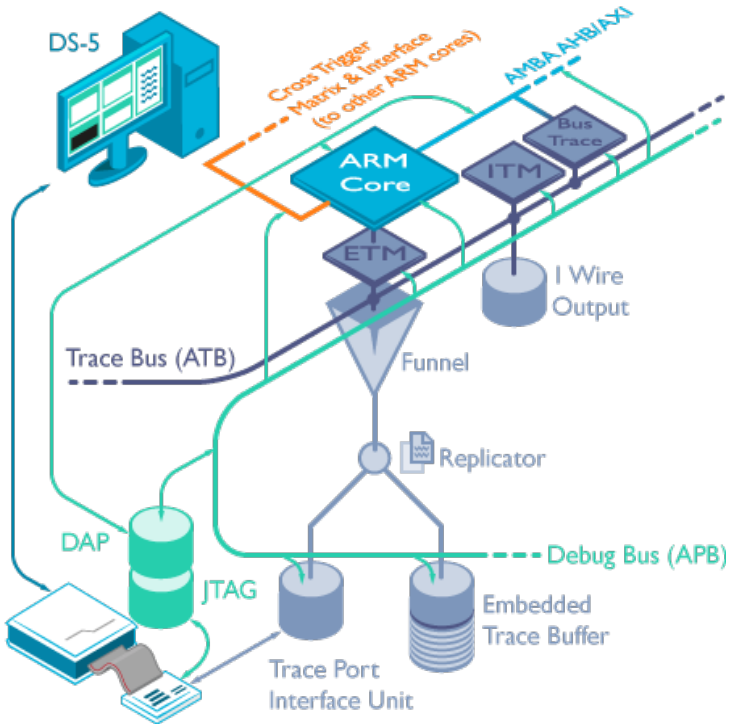


Figure 2.3: ARM CoreSight Overview [2].

In general, DAP, a subsystem used for debug and trace, is a Debug Port (DP) that is connected to one or more Access Ports (APs), providing a connection from external tools to the memory-mapped components existent in the SoC. The type of connection can be based on a simple physical interface like JTAG or Serial Wire (SW) [55, 54].

Trace information is generated by the profiling macrocells such as ETM and ITM, with key differences in their functionality. The ETM provides a real-time trace of code, data, and events of the core, by generating two trace streams of instructions and data, i.e., instruction-trace and data-trace stream. These two trace streams are then encoded into a stream of instruction-trace and data-trace packets which can be either exported off-chip to an external trace analyzer, or capture on-chip for analysis by software (self-hosted debug). The instruction trace stream contains information about instruction execution and the data-trace stream has addresses and data values of transfer that the core carries out [57, 58]. Intrinsically to the ETM are hardware functions capable of filtering the tracing information of each one of the generated streams (*ViewInst* for the instruction and *ViewData* for data stream). Both functions have a start/stop control for enabling/disabling the filtering at a particular data/instruction address. These hardware functions

also implement a include/exclude control that allows the inclusion/exclusion of a specific instruction/-data address [57]. By combining event resources that are part of the ETM (e.g., *comparators*, *counters*, *sequencers*) it is possible to trigger the start of tracing points (filtered or not) [57].

On the other hand, the ITM is a software application drive trace source, which (i) support *printf* style debugging, which is significantly faster when compared to *printf* debugging via UART, increasing the overall system's performance [59]; (ii) and trace and emits diagnostic about the system. The ITM provides a set of stimulus port registers that can be written to by the target application to output instrumentation messages to a trace sink. Moreover, the ITM unit also provides the hability to produce synchronization packets in case there are multiple trace sources and, also generate local and global timestamps to distinguish between relative (e.g., the time difference between the newer generated packed and the last) and absolute time, which is based on the present clock unit, useful for synchronization.

Other hardware resources such as the Data Watchpoint and Tracing Unit (DWT), which is key for KIR development and detailed (Section 2.1.1.5), can interact with the ITM unit to generate trace information [3, 58, 54, 47] as it is possible to see in the Figure 2.4.

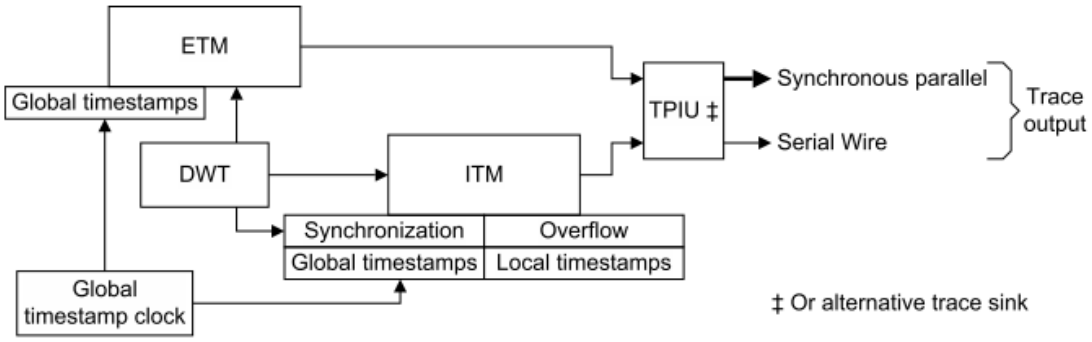


Figure 2.4: Intruction Trace Macrocell overview [3].

In case it exists more than one trace source, a Funnel, such as the CSTF, can be used to combine the multiple trace streams into a single output stream to be output into an available sink. Furthermore, a replicator can be present to duplicate the trace stream outputed and output it to different sinks that function as endpoints to store the traced streams[55]. Sinks such as the ETB are key due to the increasing difficulty of obtaining information generated from the tracing sources because of increased processor speeds. Storing the information in an on-chip buffer allows the read from it at slower rates and, therefore, removing the necessity of high-speed pads. Moreover, the core can also access the ETB by a slave-based memory-mapped peripheral, enabling it to read the trace generated by the different tracing sources

[60, 55] On the other hand, the TPIU only acts as a bridge between the on-chip trace data and a Trace Port Analyzer (TPA) [54, 55].

The connection of all the signals transmitted between the different resources present are made by the CTN, which consists of Cross Trigger Interfaces (CTIs) and Cross Trigger Matrices (CTMs). The CTIs enables the distribution of events between multiple sources such as the DWT and the ETM or trace sources and trace sinks. On the other hand, the CTIs are connected using one or more CTMs through channel interfaces [58].

Data Watchpoint and Tracing Unit: is a debug unit that provides watchpoints and system profiling for Arm's Cortex-M based processors. The unit contains several counters that can be used for calculating : (i) clock cycles (CYCCNT); (ii) folded instructions (FOLDCNT); (iii) load/store unit operations (LSUCNT); (iv) sleep cycles (SLEEPCNT); (v) interrupt overhead (EXCCNT); (vi) cycles per instruction (CPICNT). The CYCCNT is a 32-bit counter which increments for each clock cycle. The folded instruction counter has an 8-bit precision counter and increments for each instruction that executes in zero cycles. The load/store counter is also an 8-bit counter and increments for each additional cycle required to complete a multi-cycle load-store instruction and, it does not count the first cycle necessary to execute any instruction. The sleep counter is an 8-bit counter that increments for each cycle that the CPU is in a sleep mode or low-power state. The interrupt overhead counter is an 8-bit counter and increments each cycle associated with exception entry or return. Therefore, it counts the cycles related to entry stacking, return unstacking, preemption, and other exception-related operations. Last but not least, the CPICNT is also an 8-bit counter and increments each additional cycle required to execute a multi-cycle instruction, except on load/store instructions. When a counter overflows, it wraps back to 0 and continues to count. Each of the counters can be enabled and disabled individually [3, 53]. Moreover, the unit also has a set of comparators and a Program Counter Sample Register that contains the instruction address of recently used instructions. For each implemented comparator, there is an associated function register that defines its operation. The comparators can be configured to generate an event on the match of: (i) defined valued with the cycle counter value (CYCCNT); (ii) defined instruction/code address; (iii) defined data address; (iv) defined data value; (v) defined data address with a value. A match occurs, and consequently, an event, when the value specified in the comparator register is equal to the one in memory (e.g., accessed instruction address equals to the one in the comparator(s)). Accordingly to the Armv8-M architecture reference manual [3], some functions are specific to a comparator, such as the cycle counter matching function. Therefore, depending on the way the comparator function is set, the event generated can be in the form of a: (i) debug

event; (ii) data Trace match packet; (iii) data trace PC value packet; (iv) data trace data address packet; (v) data trace data value packet; (vi) ETM trigger. The debug event can cause the processor to enter a debug state (*Halting* or *Monitor* mode). In case of an *halting* debug, it is required an external tool to proceed with the core execution (e.g., step-by-step execution); on the contrary, with *monitor* debug, the core takes an exception, if enabled, where it deals with the cause of the exception. Hence, the debug event is related to the invasive type of debug, while all the other events named in the last enumeration are associated with non-invasive kinds of debug. The non-invasive types of debug from the DWT can trigger events in the ETM or output packets via the ITM [3, 53]. The relation of the comparator function with the type of event can be seen in the Table 2.1.

Comparator Function	**Debug Event**	**Data trace match packet**	**Data trace PC value packet**	**Data trace data address packet**	**Data trace data value packet**
Cycle Counter	Yes	Yes	Yes	No	No
Instruction Address	Yes	Yes	No	No	No
Data address	Yes	Yes	Yes	No	No
Data value	Yes	Yes	No	No	No
Data address with value	No	No	Yes	No	Yes

Table 2.1: Relation between comparator function and debug event. **No** means that the packet or event is not generated; **Yes** means that the packet or event is generated on a comparator match.

When watching a data address, it is possible to specify if the match should happen when it is a write, a read, or a write and a read. Another feature of this unit is the ability to link two comparators to watch a range of instruction/data addresses or an address with a specific value. When linking the comparators to watch an address range, the first comparator must be programmed with base address, while the second comparator must be programmed with the address limit. If the objective is to watch an address with a specific value, the first comparator must be programmed with the data address while the second comparator, must be compared with a value. When linking the comparators, each one can

generate different events [3]. The overall possible configurations of the comparators are demonstrated in the Table 2.2.

	Comparator Function		**Event Type**				
	First	Second	Debug Event	Data trace match packet	Data trace PC value packet	Data trace data address packet	Data trace data value packet
Instruction Address Range	Base Address	Limit Address	First	First	Second	-	-
Data Address Range	Base Address	Limit Addres	First	First	First	Second	-
Address with value	Address	Value	First, Second	First, Second, Both	First	-	-

Table 2.2: Relation between linked comparator function and event type. **First** means that the packet or event is generated by the first comparator match; **Second** means that the packet or event is generated by the second comparator match; **Both** means that a first packet is generated by a first comparator match, even if the value comparator (second) does not match, and a second packet is generated by the second comparator with the value match, if both comparators match.

2.1.1.6 TrustZone

In TrustZone-powered platforms, the processor state is determined by the address space from which it is running, i.e., the processor is in a secure state if the memory is secure; otherwise, it is in a non-secure state. The default state of the processor after reset is secure. Within the secure memory, it can be contained the code (e.g., instructions, vector table, read-only variables) and data (e.g., global variables, stack, and heap), and a non-secure callable (NSC) that contains entry functions for non-secure applications to access secure functions. To prevent non-secure applications from branching into invalid entry points, Arm introduced a new instruction named Secure Gateway (SG). When a non-secure application calls a

function from the secure world, the first instruction is the SG, and it must be placed in the NSC memory. With the placement of the SG in the NSC memory, it is prevented that any other binary data with the same opcode of the SG instruction be used as an entry point into the secure state. If a non-secure program attempts to branch into a secure region of memory without a valid entry point, it is generated a SecureFault. Moreover, if there is a call from non-secure to secure, the return to non-secure is done by the use of the BXNS instruction [4, 24].

Armv8-M TrustZone-enabled chips can also branch from secure to non-secure by using the BLXNS instruction. During state transitions, the return address present in the LR is set to a special value called FNC_RETURN. The non-secure execution can be terminated by executing a branch to the FNC_RETURN address, which triggers the unstacking of the true return address and a jump to the secure world.

The overall transitions between states and modes in a Arm Cortex-M TrustZone-enabled processor are presented in the Figure 2.5.

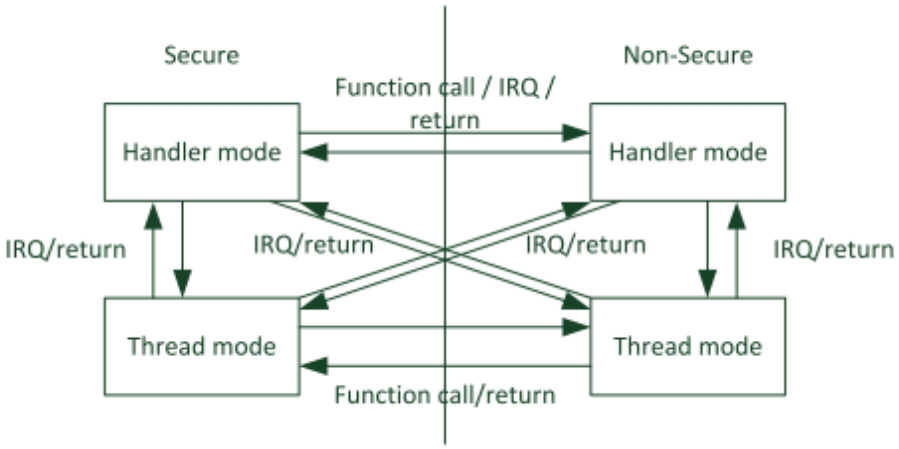


Figure 2.5: Transitions between processor's modes and security states [4].

Trusted Execution Environments. Until recently, partitioning the operating system was a pioneering way of guaranteeing local and spatial the separation between different workloads. Faults were contained in the defective partition due to memory protection, access control, and a static time-triggered scheduling scheme. To overcome the difficulties imposed by the desire of decreasing SWaP-C and integration of multiple workloads with different safety and security levels, but stil mantaining temporal and spatial isolation, new proposoal such as TEEs born in the embedded world.

TEEs deliver a secure execution environment aiming to protect the integrity and confidentiality of sensitive applications. With the use of dedicated hardware, secure-sensitive applications run inside protected

domains, isolated from the platform's OS. Solutions as Arm TrustZone [61] has been a widely used technology leveraged to implement TEEs ranging from high-end devices (e.g., Qualcomm TEE [62], SierraTEE [63]) to low-end devices (e.g., ATF-M [64], Kinibi-M [65], ProvenCore-M [66]). Trusted applications (TAs) or *trustlets* run in a secure environment guaranteed by the TEE, named secure world (SW). Therefore, TEEs are a tampering processing environment that runs on a separation kernel with the highest levels of privilege, guaranteeing the authenticity of the executed code, runtime states (e.g., CPU registers, sensitive I/O), and the confidentiality of its code, data, and runtime states stored on a persistent memory [5] (see Figure 2.6). Therefore, TEEs enable modern devices to provide an enormous range of functionalities while still meeting the running applications' requirements and ensuring their data privacy and isolation using secure-base infrastructures (e.g., Arm TrustZone) [67].

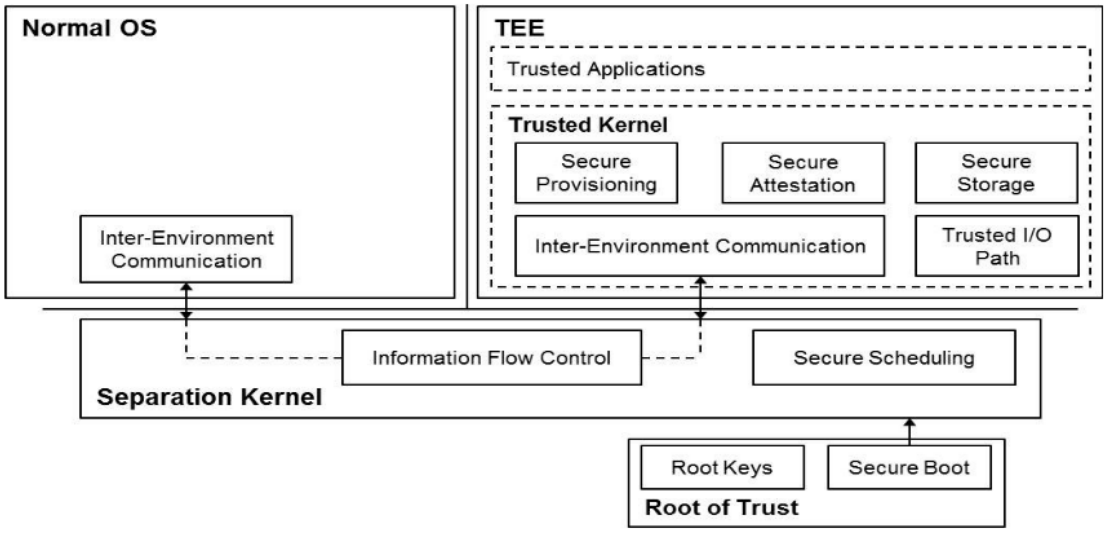


Figure 2.6: Trusted Execution Environment building blocks [5].

2.1.2 uTango TEE

Over the past years, TrustZone-assisted TEEs targeting Cortex-A processors, have been attacked multiple times, producing results that show several architectural deficiencies, critical implementation bugs and overlooked hardware properties [67]. At the architectural level, it was highlighted the (i) excessively large trusted computing base (TCB); (ii) large number of interfaces; (iii) existence of multiple privileged secure kernel drivers; (iv) asymmetrical isolation between worlds [6]. Although the problems faced are related to commercial implementations focusing Cortex-A processors, newer solutions proposed for TrustZone-M

microcontrollers (e.g., ATF-M [64], Kinibi-M [65]) are also failing in the same fields as their predecessors. Moreover, Arm is spreading ambiguous messages regarding what should be deployed in the secure world, leading to conclusions that the current proposed dual-world model is not enough to the increasing complexity of modern IoT devices. Therefore, it was recently proposed a novel multi-world architecture, named uTango, enabling the execution of multiple environments with strong isolation between the multiple workloads [6].

uTango is based on the zero-trust model, which dictates that every single component, except the TEE kernel, cannot be trusted. Thus, multiple applications, services, or workloads are combined into equally secure and isolated domains running in the non-secure world (NW), named Non-Secure Virtual Worlds (NSVW). uTango TEE is built on three fundamental principles: (i) principle of least privileged where the entire containment of the TEE to attacks rely on hardware support as much as possible; (ii) principle of minimal implementation where uTango kernel has the highest privileged level and secure applications/services of the normal world are de-privileged, having only access to essential resources (e.g., devices, system services); (iii) principle of containment that defines clear boundaries of all execution domains, so that they are limited to their own resources.

2.1.2.1 Architecture Overview

uTango kernel is the only component running on the secure-world (SW) with the highest level of privilege (i.e., secure handler mode), while all NSVW run in the NW. uTango aims to a clean and minimal implementation and, therefore, the kernel is composed by three main components: (i) system partitioner (SP) that relies on a file that defines each NSVW boundary and resources; (ii) worlds scheduler; (iii) and worlds' communication channel (WCC). The first piece of uTango workflow starts with the configuration of each NSVW by setting its: (i) application id; (ii) SP initial location; (iii) application Reset Handler location; (iv) VTOR memory address; (v) number of SAU regions, where it is set the start and size of each region; (vi) number of IRQs and their number in the vector table.

The partition of each one of the NSVW is assured by the usage of TrustZone-assisted hardware existent in the Armv8-M platforms (e.g., SAU). Each world's configuration state is saved into a data structure named world's control block (WCB) (i.e., CPU register bank, SAU configuration table, selective System Control Block (SCB) registers, and interrupts table). Regarding the security gates, the system partitioner does a one-time setup. Furthermore, uTango assures that all accesses and transitions to all the bus masters are trapped and mediated, preventing a possible reconfiguration of each bus filter during a context-switch.

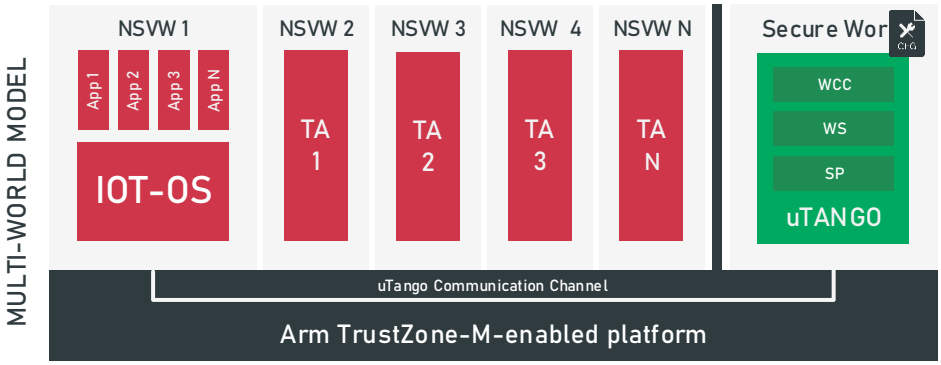


Figure 2.7: uTango multi-world architecture [6].

uTango imposes temporal isolation by the usage of a scheduler supported by an architectural time unit (e.g., Arm SysTick). For each tick, each NSVW is scheduled using the round-robin model. At each scheduling point, it is (i) saved the suspended NSVW context into the WCB; (ii) scheduled a new NSVW to be resumed; (iii) reconfiguration of the SAU regions accordingly to the NSVW's WCB; (iv) and restored the context of the new NSVW. Moreover, uTango offers a communication channel in the form of four application programing interfaces (API), that allows the exchange of *full duplex* (i.e., send and receive) secure 12-byte messages between NSVWs, in a blocking or non-blocking way. The WCC acts as a messaging gateway that receives each sent message and forwards it to the respective NSVWs. The system designer is responsible for defining the messages and semantics for each target application [6]. An example of a uTango multi-world architecture is presented in the Figure 2.7.

2.1.2.2 Discussion

As preivously explained, embedded systems distinguish themselves for the limited availability of resources and the requirement to satisfy non-functional constraints such as low latencies and high throughputs. In several application domains such as automotive, avionics, or industrial automation, many of the system functionalities are associated with strict requirements on deadlines to deliver results of calculations. The failure to meet deadlines may cause catastrophic or at least highly undesirable system failures, associated with risks for human or economic damages [68, 69]. The newer designed real-time applications demand higher performant systems but at the same time guarantee tight worst-case execution time bounds. Often the execution time of each real-time task is overestimated to ensure deadlines are met. Consequently, the whole system's performance is affected, and the costs increase [70].

With the aim of tackling the barries that arise from the different microarchitectural deficiencies created to increase the systems' overall performance and, at with the craving to test KIR in a realistic scenario,

the mechanism will be integrated into a TEE. From the multiple available solutions (e.g., Kinibi-M [65], ProvenCore-M [66]), uTango was chosen since it is an open-source solution that fulfills all the requirements. In addition, since it is in-house developed, it came easier for its full understanding and usage.

2.2 Related Work

This section is dedicated to analyzing the existing solutions proposed by the scientific community focused on increasing the system's predictability. Due to a wide specter of proposed solutions, only a subset of works is presented and organized by the type of solution. Hence, this section is divided into three different subsections differentiated by the solution applicability: (i) cache level reservation by software; (ii) memory bandwidth reservation by software; (iii) hardware memory infrastructures.

2.2.1 Contention-Aware Cache Partition

This subsection presents four different solutions capable of increasing the system's determinism by partitioning the last level of cache, which is typically shared between cores in last-generation multi-core platforms. The techniques used are dependent on the available hardware facilities and can go from cache coloring to cache lockdown. Most of the proposed solutions use a separation kernel (e.g., hypervisor) to enforce spatial and temporal isolation between the workloads.

2.2.1.1 Deterministic Memory Hierarchy and Virtualization for Modern Multi-core Embedded Systems

In this paper, Bertogna et al. [38] presents a software-based framework to restore memory access determinism in high-performance embedded systems. The developed work is based on the premise that modern multi-core embedded SoC provides the necessary hardware facilities to allow cache manipulation.

The work proposes the implementation of memory coloring at the hypervisor level, allowing the partition of the last-level cache among different guest OS, while requiring no modification to OS-level memory allocators and/or to the guest OS's toolchain. By carefully performing the allocation of physical addresses to user-space applications, it was possible to partition access to resources in the shared memory hierarchy. The technique used is called page coloring and allows the temporal isolation between memory-intensive applications on different cores. Each core is assigned a set of colors that are inherited by the running

applications. Typically, depending on the virtualization extensions existent in modern SoCs, two different implementations of page coloring are possible.

In the first method, systems with one-stage address translation, the OS should be modified to ensure that applications are allocated to physical memory from a different pool of pages depending on the core, they will execute. On the other hand, the second method consists of two-stage translation, and the hypervisor is responsible for mapping intermediate physical addresses of different OS to physical addresses with non-overlapping colors. Both approaches present disadvantages due to the necessity of modifying the OS's or changing the hypervisor code to specific SoCs. Thus, they propose an alternative approach called *Boot-first*, *Virtualize-later*, where they boot the system via an unmodified OS (i.e., called root partition).

For this scenario the used hypervisor (Jailhouse) [71] dynamically virtualizes the root partition, activates the two-stage virtual address translation, and provides an interface to define and enable additional partitions. By also removing virtual CPU scheduling, the hypervisor is kept to its bare minimum, simplifying its code-base and enabling quicker porting to different SoCs and its maintainability.

Each partition in Jailhouse is associated to a set of memory region descriptors linked to a color, defining how physical address memory will need to be mapped to intermediate physical addresses. During boot, physical addresses are re-mapped to color-compliant physical addresses by the Jailhouse hypervisor (a technique named dynamic re-coloring). To achieve the color re-mapping, first, the root partition execution must be suspended, then a newer physical address must be computed. Next, the old data located in the old physical addresses are copy to the colored addresses. Moreover, the second-stage page tables must also be updated to map old intermediate physical addresses to colored ones. To finish, the cache and translation lookaside buffers (TLB) are flushed. Once all is done, access to an old address will result in the access to a colored physical address.

The authors also investigated the impact of a set of traditionally neglected hardware features that have a significant effect on predictability (hardware prefetchers, two-stage address translation, and self-eviction due to pseudo-random replacement policy). Based on the study, the authors presented a novel technique named IDA, invalidation-driven allocation, to deterministically control the content of a random replacement policy for applications with a small memory footprint. Usually, the cache replacement policy is invoked to select a line to evict only if none of the selected set lines is invalid. IDA is based on the inverse; if an invalid line exists, it will be deterministically selected without evicting any other line. This ensures that new cache lines accessed by a running task will not overwrite one with another.

The proposed solutions are evalueted in four different stages: (i) coloring only; (ii) virtualization coloring; (iii) IDA; (iv) integrated framework. In the first stage, two interference cells run an application that pollutes the shared cache. Without cache coloring, the two cells make memory increase the overall latency of accesses while under coloring shows a significant decrease in the latency time. Moreover, in the second stage, i.e., under virtualization coloring, it is compared the latency memory accesses in cacheable or non-cacheable regions, when the hypervisor is enabled or disabled and if the access causes a TLB miss or hit. Results show that accesses to cache or non-cached data present in the TLB (i.e., TLB hit) are not affected by the presence of the hypervisor, but increase when a miss occurs because translation must undergo a two-stage of page table walk.

To evaluate IDA, it was used a set of benchmarks, i.e., synthetic and realistic. Under the tests, it was considered the cases where prefetching was enabled or not. When IDA was used without the system prefetchers, it outperformed the legacy cases (i.e., no mechanism) by maintaining the access times near-constant. Moreover, by combining prefetchers and IDA, the access times further dropped, showing better results than using the prefetcher only.

Lastly, it was tested all of the proposed solutions and measured the overall performance and predictability gains when subjected to realistic scenarios. Results show when coloring alone is not enough to solve contention problems, but IDA and coloring increase the overall system predictability, leading to the cache miss per line very near to a theoretical value of 1 in the LLC.

2.2.1.2 vCat: Dynamic Cache Management using CAT Virtualization

vCat is a novel design based on Intel's Cache Allocation Technology (CAT) for dynamic shared cache management on multi-core virtualized platforms [7]. The mechanism aims to deliver strong shared cache isolation at both the virtual machine (VM) and task levels, being able to be configured both statically and dynamically. The prototype was developed on top of Xen [72] hypervisor and the real-time Linux based extension LITMUSrt [73].

Intel's CAT is a hardware feature that allows the control of the shared last-level cache allocation to the physical cores. CAT divides the shared cache into N non-overlapped equal-size cache partitions. A set from these cache partitions can be allocated to a CPU. CAT also ensures that the VMs and tasks can only access cache lines in the newly assigned partitions and not in the older ones, in cases of dynamic cache allocations.

Since CAT only provides core-level cache isolation, this technique aims to achieve hypervisor and VM level cache allocations based on the CAT hardware available. The partitions of the cache are made at the hypervisor level, where physical memory is turned into "virtual memory." Then each one of the partitions is allocated to each one of the VMs.

The used technique also allows partitions to be preempted, allowing partitions to be shared between VMs. If so, at the guest level, the kernel must dynamically allocate memory for its tasks when they are scheduled in case they are preempted. Furthermore, partitions are allocated to real-time tasks based on either a first-come-first-served or criticality. The tasks with the least priority, i.e., best effort, are given unallocated partitions as presented in Figure 2.8.

For testing the proposed solution, the authors used the PARSEC benchmark and synthetic workloads. The goal was to evaluate how vCAT could (i) diminish the task-level cache interference between multiple tasks; (ii) improve the system's real-time performance when using static and dynamic cache allocation.

The experimental results proved that at the task-level, vCAT provided isolation could effectively avoid WCET slowdowns caused by cache interference. Furthermore, both static and dynamic cache management improved the schedulability of the tasks substantially. Between the two cache management schemes, the dynamic one outperformed the static scheme, which was expected since it is more effective in handling workloads with dynamic timing constraints.

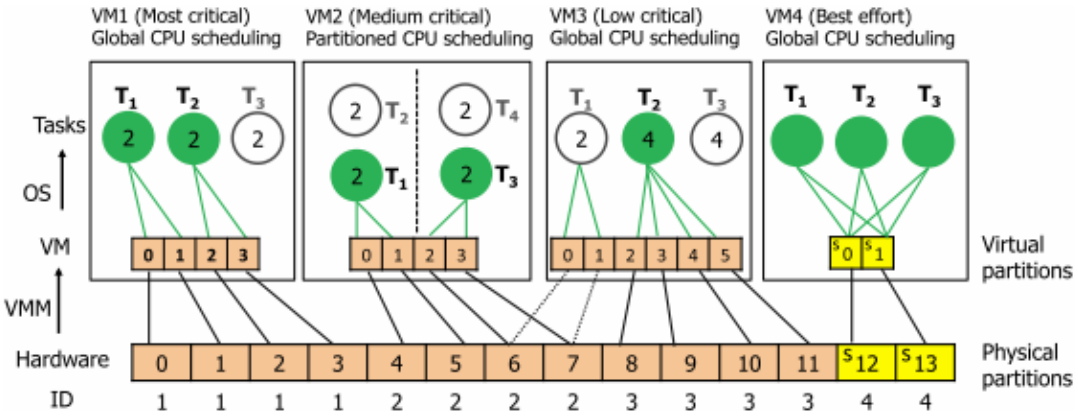


Figure 2.8: Dynamic cache management with vCAT [7].

2.2.2 Contention-Aware Memory Bandwidth Reservation

This subsection focus on contention-aware solutions that leverage specialized hardware performance counters to manage and restrict memory usage of workloads according to a defined memory bandwidth

budget. Performance hardware counters are currently available in several last-generation multi-core platforms across a variety of architectures, from Intel's Performance Monitoring Counter (PMC) [74] to the Arm's Performance Monitoring Unit (PMU) [75]. Moreover, these solutions are typically tightly-coupled with the kernel's scheduler.

2.2.2.1 Supporting Temporal And Spatial Isolation in a Hypervisor for ARM Multicore Platforms

The authors of this work [8] propose a two-stage approach to increase predictability at the last level cache (LLC) and the DRAM controller in this work. The XVISOR [76] hypervisor is used to integrate the solution, which is deployed in a Raspberry Pi 2 platform.

The proposed solution consists of using cache coloring to partition the last level of cache. A set of colors is set to each VM, which are defined during the configuration phase. At runtime, when the hypervisor configures the second stage of the Memory Management Unit (MMU), memory pages are mapped into the memory areas whose addresses match the VM domain's color. By associating different non-overlapping colors to other domains, they guaranteed isolation in the last level of cache. Hence, the resulting effect is that each domain will have its own LLC. Figure 2.9 shows an example of a colored memory map cache for two domains, each assigned four different colors.

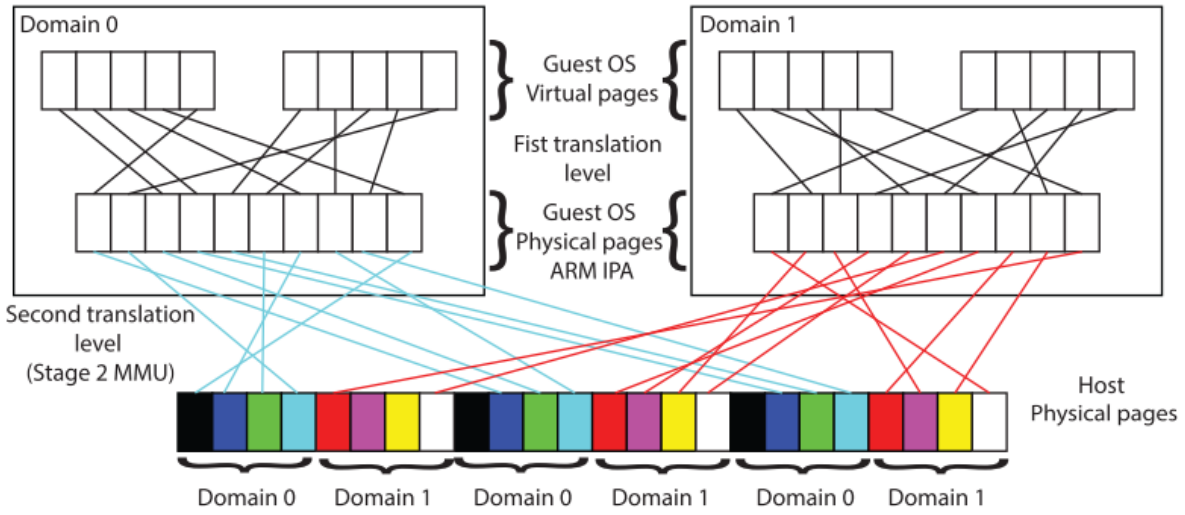


Figure 2.9: Cache coloring example [8].

In the second stage, the system uses a memory reservation system where each domain has an assigned budget of memory accesses for each VM (i.e., a virtual CPU (VCPU)). After a period of time, the

budget is replenished. To know how much bandwidth the VCPU has consumed, the authors used the PMU available in each of the cores configured to trigger an interrupt once a predefined value is exceeded.

Each one of the domains running in XVISOR is preempted by a modified scheduler with an integrated memory reservation system. At scheduling points, the proposed mechanism evaluates the domains that can transition from *READY* state to *RUNNING*. In case of PMU interrupt overflow, the VM transitions to a *RECHARGE* state and the VCPU is suspended, not being eligible for execution until the budget is replenished. The overall VCPU state machine of the memory bandwidth reservation system is presented in Figure 2.10.

To test the system, the authors created two domains (i.e., 0 and 1) and assigned each one to a dedicated core. Domain 0 executes a variant of Isol-Bench benchmark suite [77] and, Domain 1 was configured to continuously access a large portion of memory, i.e., main interference of the system. Under cache coloring, the system, when compared to the results without cache coloring, showed an improvements due to the fewer cache evictions caused by domain 1. Moreover, to test the bandwidth reservation mechanism, Domain 1 was limited to a range of bandwidth limits. As the bandwidth of Domain 1 decreased, the overall predictability of the system increases. Therefore, the experimental results proved that the implemented solutions are highly effective and, consequently, the running time of the benchmark is significantly reduced.

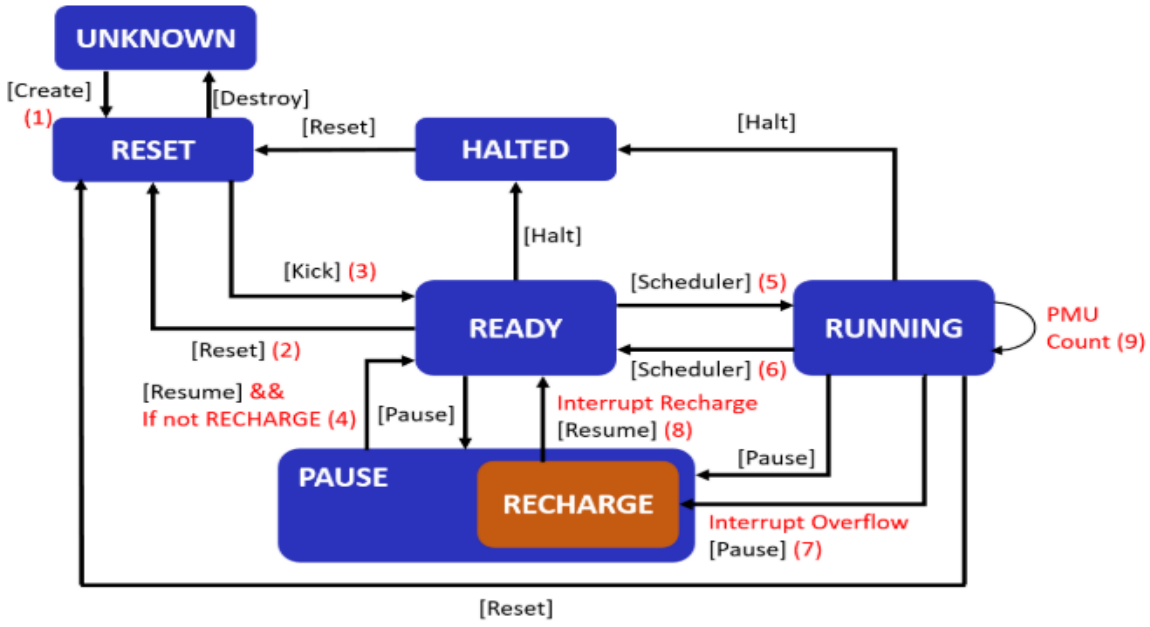


Figure 2.10: VCPU state machine with memory bandwidth reservation mechanism [8].

2.2.2.2 MemGuard

MemGuard [9] is a memory bandwidth reservation system for efficient performance isolation in Intel multicore platforms. The solution mainly focuses on soft real-time systems [9] and it distinguishes memory bandwidth into two parts (i.e., *guaranteed* and *best effort*). MemGuard provides bandwidth reservation for the *guaranteed* bandwidth for temporal isolation, and with efficient reclaiming so that the reserved bandwidth usage is maximized. The *guaranteed* bandwidth represents the minimum service rate the DRAM system can provide, while the additionally available bandwidth is the *best effort* and can not be guaranteed by the system.

The memory bandwidth reservation is based on the *guaranteed* part to achieve temporal isolation. However, to increase the system's efficiency and guarantee that the all-time slot is all used, a reclaiming mechanism is introduced. Based on each core's usage prediction of memory, the core can donate back bandwidth so another core can claim that. Since the reclaiming algorithm is prediction-based, mispredictions can lead to situations where *guaranteed* bandwidth is not delivered to the core. The throughput of the system is further improved by exploiting the *best effort* bandwidth after the *guaranteed* of each core is satisfied. Moreover, when resource usage changes over time, a static reservation approach results in poor resource utilization and poor performance. If the maximum bandwidth is attributed to a task, it can result in waste since it would not use it entirely. By giving bandwidth to each core, dynamically, each of the cores would have its requirements fulfilled when needed. Thus, MemGuard is compoased by two main software components: (i) the *per-core regulator*; (ii) the *reclaim manager*.

The *per-core regulator* is responsible for monitoring and enforcing the core's bandwidth usage limits. To keep track of the memory usage, the mechanism reads information from the hardware PMCs. When the memory usage reaches a pre-defined threshold, it is triggered an overflow interrupt. Each one of the regulators has a history-based memory usage predictor. Each regulator can donate its budget to the *reclaim manager*, based on the predictor; thereby, cores can start reclaiming once they used up their entire budget. The *reclaim manager* receives the donated extra bandwidth from each core and has the function to redistribute it to the regulators when requested. The overall system architecture of MemGuard can be seen in Figure 2.11.

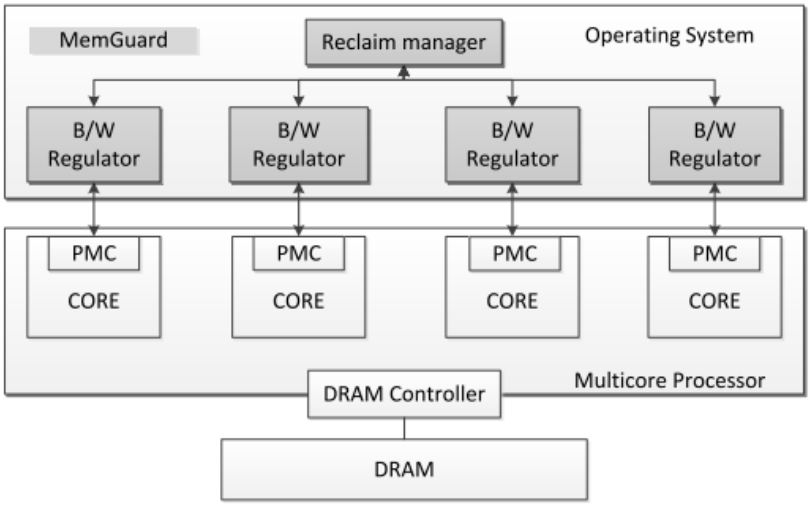


Figure 2.11: MemGuard system architecture [9].

MemGuard provides two levels of memory bandwidth reservation: *System-wide reservation* regulates the total allowed memory bandwidth such that it does not exceed the minimum DRAM service rate and, *per-core reservation* assigns a fraction of the minimum DRAM service rate to each core. For each execution period, each core has an assigned budget. It is essential to notice that MemGuard is integrated into a modified Linux kernel, and in this case, the period window equals each tick of the scheduler, which is better for predictability, but worse for the system performance due to the interrupt and scheduling overhead. The system starts by resetting the PMC and, when the interrupt is received, the regulator calls the OS scheduler to dequeue all tasks from the core's run-queue so that the core can not access any more memory until the next period starts. At the beginning of the next period, the budget is replenished in full, and all dequeued tasks, if any, are queued to the run-queue.

Since *system-wide reservation* only accounts up to the minimum service rate (i.e., guaranteed bandwidth), the authors tried to exploit the spare memory bandwidth exceeding (best-effort bandwidth) whenever it is possible, increasing the overall system throughput. When all cores use their assigned budgets, the remaining time until the new period begins is considered spare, as it is shown in Figure 2.12. MemGuard tries to maximize memory by allowing cores to continue its execution. This is only permitted after all cores have depleted their budgets so that cores running within the budget do not suffer from intensive memory contention.

Under SPEC2006 benchmarks[78], Memguard proves to be able an efficient mechanism under heavy memory-intensive workloads. By using the reclaiming algorithm, the solution also showed improvements

in the overall system throughput, when compared to a reservation-only system.

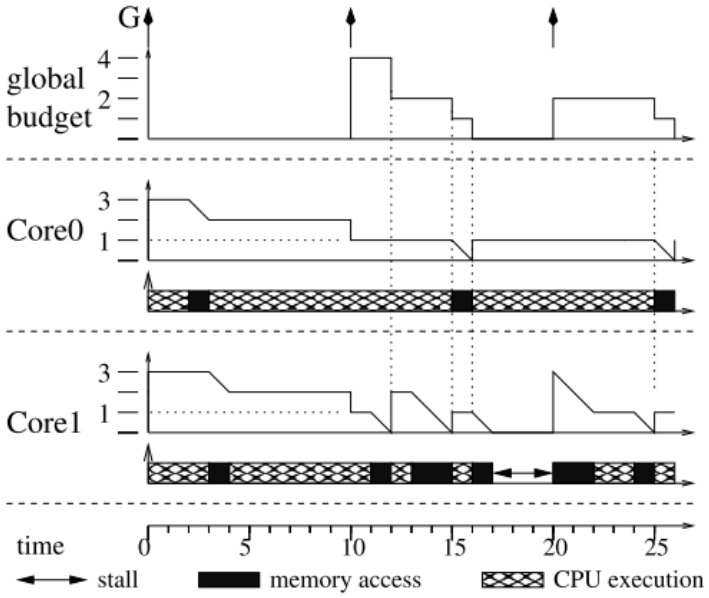


Figure 2.12: MemGuard illustrative example with two cores [9].

2.2.3 Contention-Aware Hardware Memory Infrastructure

This last subsection presents a review of works that focus on different aspects of modern multi-core devices' memory infrastructure. The first work, i.e., PALLOC [34], proposes a memory allocator that partitions the application's memory pages to specific DRAM banks. Next, Ringhofer et al. [35] propose a novel memory controller design that guarantees minimum bandwidth and a maximum latency bound. The last segments in this subsection present two studies regarding the impact of multi-core contention on shared memory resources (e.g., caches, prefetchers, memories) in a last-generation high-end platform Xilinx Ultrascale+ Multi-Processor System-on-a-Chip (MPSoC) and a low-end MCU (Arm Musca-A1).

2.2.3.1 PALLOC

PALLOC is a DRAM bank-aware kernel memory allocator for performance isolation on multicore platforms [34]. The mechanism is implemented in a Linux OS and fully compatible with existing COTS hardware platforms and transparent to applications, removing the need to modify their source code. PALLOC consists of exploiting the page-based virtual memory system to allocate memory pages on specific DRAM

banks, diminishing bank sharing among concurrently executing applications. The proposed solution allows a system designer to flexibly partition DRAM banks to improve the quality of performance isolation of a multicore platform, which means each core has its own private DRAM bank. Partitioning the banks eliminates bank sharing without the need for hardware modification. Although the partitioning of the DRAM banks between the cores can lead to no contention, it has the disadvantage of not maximizing the available banks' usage, i.e., not use more banks even if they are free. Therefore, possible contention is limited only to data bus sharing.

By performing a set of SPEC2006 benchmarks [78], results show that creating private DRAM banks for each core improves performance isolation and real-time performance on multicore platforms. Moreover, the authors concluded that partitioning DRAM banks is not ideal for performance isolation due to contention in other possible shared resources (e.g., memory bus, memory hierarchy).

2.2.3.2 Evaluating the Memory Subsystem of a Configurable Heterogeneous MPSoC

Caccamo et al. [10] evaluated the memory subsystem of the Xilinx Ultrascale+ MPSoC, to study the impact of multi-core contention on micro-architectural shared resources.

Focusing on strong temporal isolation among high-performance cores on the considered heterogeneous MPSoC, the authors proposed the software architecture shown in Figure 2.13. Jailhouse hypervisor [71] is used as a separation layer between the running OSs and the underlying hardware. Non-critical tasks run in a Linux general-purpose OS, while safety-critical tasks are delegated to the Erika RTOS [79]. The design prohibits access to shared resources from different cores, removing unbounded contention and avoiding system unpredictabilities. Moreover, page coloring is also used in the LLC with the support of Jailhouse.

Experimental results show that the primary sources of contention derive from shared resources such as the LLC and main memory used by the processing system (PS), such as DRAM. The page coloring of the last level of cache removed part of the contention, while at the DRAM level it was introduced a DRAM bank-aware memory allocator (PALLOC) [34]. With cache partitioning and the PALLOC mechanism, a specific amount of cache and dedicated DRAM banks were assigned to a particular core, enforcing strong isolation between the OSes running on different cores. To finish, the authors also implemented their own block of ram in the programmable logic of the Ultrascale+, namely a BRAM. The introduced BRAM does not show less latency between serialized and random access; thereby, it is independent of the incoming traffic, i.e., contention-free when accessed by multiple cores.

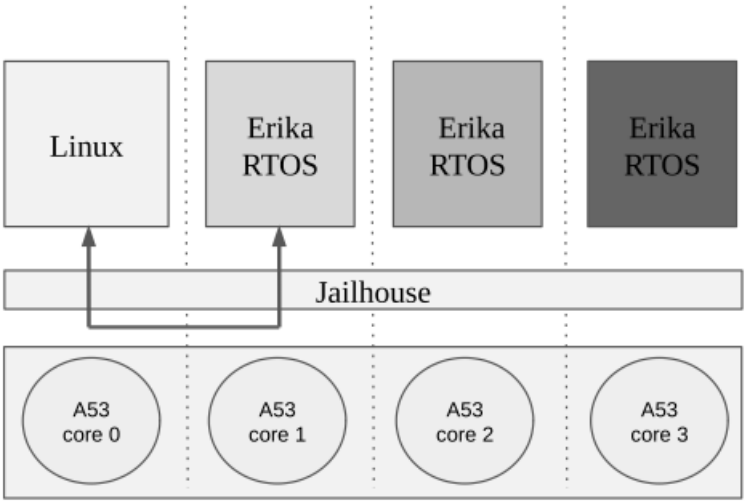


Figure 2.13: Overview of the proposed software architecture [10].

2.2.3.3 Virtualization on TrustZone-enabled Microcontrollers? Voilà!

Sandro et al. [11] aimed to develop a lightweight timing predictable infrastructure, i.e, an hypervisor for TrustZone-enabled microcontrollers. The experiments were conducted on the Arm Musca-A multi-core platform based on the Armv8-M architecture. It encompasses the CoreLink SSE-200 subsystem featuring an asymmetric dual-core Cortex-M33 with 2KB independent instruction cache for each one. The cores are connected to the main bus matrix layer (i.e., multi-layer AHB5 interconnect), enabling parallel access paths between multiple masters and slaves. Connected to the main bus are low-size internal SRAMs (iSRAM). Moreover, one of the slave ports connected to the main bus allows the cores to access external memory elements used to store code (e.g., external SRAM (eSRAM) and QSPI boot flash memory). Therefore, full concurrency access from both cores to different external code memories is not possible without contention.

The implemented hypervisor targets a dual-OS configuration where the physical core is virtualized into two by using TrustZone technology, i.e., secure and non-secure. The scheduler implements an asymmetric scheduling policy where the non-secure VM only executes during the secure VM's idle periods. The secure VM resumes execution as soon as an event occurs (e.g., interrupt), as shown in Figure 2.14. The non-secure interrupts are configured to have a lower priority than the secure interrupts and are disabled while the secure VM is executing. Therefore, it is avoided any kind of denial-of-service (DoS) attacks. In each context-switch, the CPU registers (R0-R12, LR, PC) are saved into a software Virtual Machine Control Block (VMCB). On TrustZone-M-based systems, the memory space can be partitioned according to the attributes of two units: Under TrustZone-M, the system memory space is partitioned used the SAU and IDAU. The

security of the memory regions (e.g., Secure, Non-secure, Non-secure Callable) are determined by a logic *OR* operation between the IDAU and SAU. Furthermore, the MPC (i.e., *security gates*) determine the overall systems' security (e.g., system bus, TrustZone-oblivious slaves).

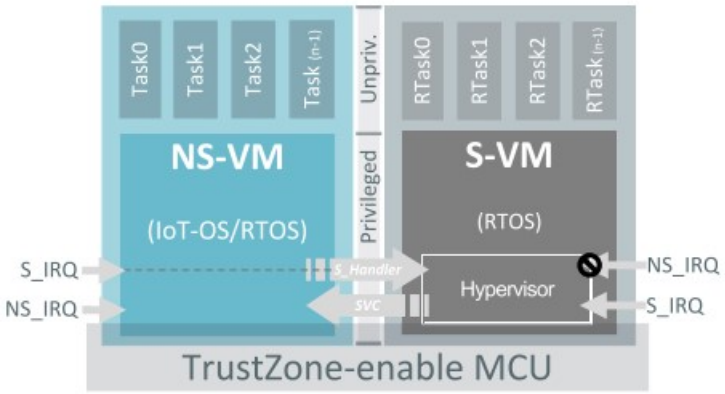


Figure 2.14: TrustZone-M-assisted hypervisor (single-core) [11].

In addition, due to the platforms shifting towards multi-core, the proposed solution was extended to multi-core in an asymmetric multi-core (AMP) configuration. In this configuration, each VM is pinned to a specific core, i.e., secure VM (S-VM) is assigned to the main processor (CPU0), while the non-secure VM (NS-VM) is assigned to the secondary core (CPU1) as demonstrated in Figure 2.15. The hypervisor is split into two parts, where the master runs in the CPU0 and the slave in the CPU1. The master hypervisor is responsible for the memory partition, interrupts configuration, and kickoff of the secondary processor. The slave hypervisor partitions and configures the interrupts accordingly to the master hypervisor directives.

To achieve a contention-free memory layout for an AMP configuration, the authors proposed the use of the iSRAM0 by the S-VM and the iSRAM3 to the NS-VM. Due to the natural size of FreeRTOS, parts of the code were migrated to the eSRAM. Therefore, eSRAM by both VMs and independent iSRAMs for data. Although the eSRAMs are shared, the instruction caches' use decreased significantly the existent contention in the system. The proposed solution, although functional, leads to possible attacks by forcingly thrashing the cache lines and complicated *ad-hoc* modified linkerfiles. With the proposed memory layouts, the experimental results demonstrated that the system is highly predictable and with high efficiency.

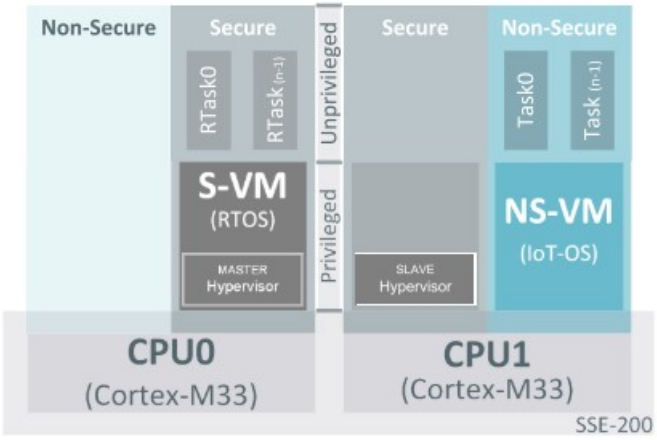


Figure 2.15: Asymmetric multi-core system architecture (AMP) [11].

Chapter 3

System Specification

The previous chapter exposed the main concepts and related work that support this dissertation. Consequently, this chapter aims to specify the architecture of the KIR system (Section 3.3) and establish the experimental setup (Section 3.4) of the final evaluation to be performed. Before designing the KIR mechanism, a preliminary contention analysis has been performed on the selected Arm's Musca-A1 platform. The drawn conclusions of such analysis are detailed in Section 3.2.3. Lastly, Section 3.1 presents relevant aspects surrounding the development environment and the selected platform.

3.1 Platform and Tools

This subsection aims to describe the platform and tools to be used for the testing and development of the proposed mechanism. The chosen platform required to fulfill the following requirements for the to study take place: (i) a multi-core architecture; (ii) hardware shared resources (e.g., at bus level, memory or cache); (iii) ways of measuring resource consumption within the system at runtime; (iv) uTango TEE support; (v) available and well-documented open-source tools and resources supporting the platform. Moreover, this subsection encompasses a set of benchmark analyses used to deeply understand the platform's contention.

3.1.1 ARM Musca-A1

Arms's Musca-A1 board implements the CoreLink SSE-200 subsystem [12] which includes: (i) two Cortex-M33 TrustZone-enabled cores; (ii) 2 KB of non-shared instruction cache for each core; (iii) four internal SRAM (iSRAMs) of 32 KBytes of size each; (iv) four external code expansion SRAM (eSRAMs) of 512 KBytes of size each [80]; and (v) 8MB of QPSI Flash external to the CoreLink subsystem. The

memories can be configured as secure or non-secure, a feature enabled by the TrustZone infrastructure. Both secure and non-secure represent the same memory space but have different aliases associated with them accordingly to their security state. The iSRAMs can be accessed for data and code, while the remaining memories, i.e., Flash, and eSRAMs, are code expansions and can only be accessed for code.

The implemented instruction cache reduces the code access fetches targeting the Flash memory and eSRAMs memories. Moreover, the instruction cache reduces the overall latency of code access by having zero-cycle access time on address hits. On the other hand, a cache miss causes a fetch and, therefore, an extra bus latency cycle to access the requested data. Thus, subsequent operations are stalled while the fetch process occurs.

Each cache includes a configuration interface that is only accessible to the processor that is connected to it. The cache has a bit named HALLOC in the Instruction Cache Control register (ICCTRL) that enabled the handler allocation. If set to 0, all incoming handler code fetches do not allocate a cache line if a miss occurs. If the fetch results in a cache hit, the access is treated as if it is cached. This allows handler code accesses to be more deterministic and avoids cache trashing if there are too many interrupts. Otherwise, if set to 1, the handler code access is treated like any other code access arriving at its interface.

The instruction cache also offers other functionalities such as gather statistics(e.g., cache misses or hits) and manipulates its contents by setting specific bits of the ICCTRL register. The cache statistics are only gathered if the STATEN bit is set and, the corrosponded values can be read from specific registers (e.g., instruction cache statistic hit (ICSH) register, instruction cache statistics miss count (ICSM) register). To clean the statistics, it is required to change the STATC bit to 1. Moreover, the cache is only enabled if set the CACHEEN bit to 1, and its lines can be fully invalidated if the FINV bit is set to 1 [80].

In accordance with the manuals [80, 81, 12, 56] the board implements an ETM, a ITM module, and an DWT unit for each core. The DWT, besides the counters previously mentioned (section 2.1.1.5), has four comparators. All of the comparators can be configured to watch an instruction or data address. Only the comparator 0 may be used for comparison with the CYCNT counter, and only comparator 1 and 3 can be used as instruction or data address limit, which means that they can be linked with other comparators to watch a range of addresses or data address with a value (e.g., comparator 2 and 3) [81].

Regarding the multi-core system, only the CPU0 boots at reset. To wake up the second core, CPU0 must configure the VTOR register in CPU1. Moreover, the SSE-200 subsystem offers a message handling unit (MHU) with multiple channels to transmit messages between cores, enabling a inter-core communication. The MHU unit also provides an interrupt, which, if enable, is capable of interrupting the core when

a message is received [80].

The Musca-A1 test board also implements a set of 32-bit timers habilitated to generate interrupts, and two UARTs targetting the secure or non-secure state (Figure 3.1) [80, 12].

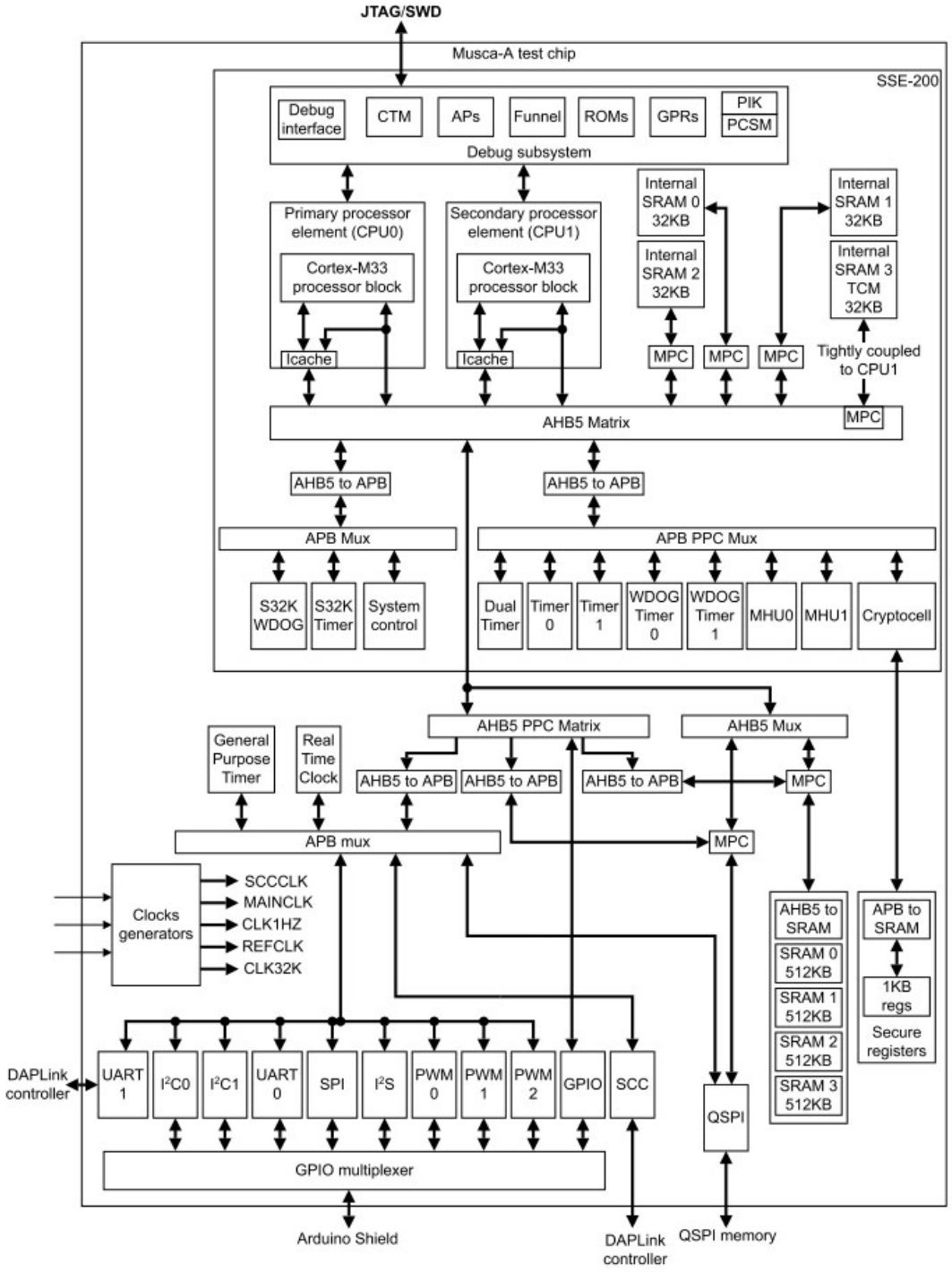


Figure 3.1: Architecture of the Musca-A test chip [12].

3.1.2 Development Environment

The development environment will allow the implementation of the proposed mechanism. The environment encompasses both hardware and software platforms that will be specified throughout this subsection.

To set up the development environment, it was used a set of open-source tools provided by Arm and GNU. These tools were chosen based on their accessibility in different platforms and their support by the uTango TEE. For compilation and debugging the GNU Arm Embedded Toolchain was selected, which contains integrated and validated packages featuring the GCC compiler, libraries, and other tools necessary for bare-metal software development [82].

For automation of the buildsystem, the GNU make tool was used [83]. Moreover, to debug in-place the target platform, i.e., create a server-bridge between the host and the target, the PyOCD was used [84]. Lastly, to ease the debugging task, the GNU MCU Eclipse [85] was selected to provide a graphical view.

3.1.3 Benchmarking Suite

There are a variety of benchmarks that mostly focus on either the CPU or the memory subsystem. From a numerous list of available benchmarks, the possible choices were narrowed down to only four that would be optimal for the work to be developed. The first one is the STREAM synthetic [86] benchmark designed to measure sustainable memory bandwidth (in MB/s); however, the benchmark focus on high-performance computer and required large amounts of memory, which is not the case of the Musca-A1 platform.

Another benchmark is EEMBC Autobench that consists in a set of benchmarks that allows users to predict the performance of microprocessors and microcontrollers in automotive, industrial and general-purpose applications [87]. It includes 16 benchmarks kernels that range from (i) generic workload tests that include bit manipulation, matrix mapping, encryption algorithms, etc; (ii) basic automotive algorithms that include controller area network (CAN), angle-to-time conversion, road speed calculation, etc; (iii) and signal processing algorithms such as fast fourier transforms (FFT), finite impulse response filter (FIR), etc.

Another potential bencharmark was Multibench [88], that consists of a suite of benchmarks that allows the processor and system designer to analyze, test, and improve multicore processors by using three forms of concurrency: (i) by data decomposition in multiple threads cooperating on achieving a unified goal; (ii) by processing multiple data streams by using common code running over multiple threads; (iii) by using multiple workload processing that focuses on concurrency at data and code level.

Lastly, it is Coremark [89] which is a synthetic benchmark that measures the performance of the CPU. It consists of algorithms such as list processing (find and sort), matrix manipulation, state machine to determine if an input stream contains valid numbers, and cyclic redundancy check (CRC). To ensure that compilers cannot pre-compute the results it is used a set of values that are not available at compile time.

3.2 Preliminary Contention Analysis

This subsection presents a detailed analysis of a modern TrustZone-M-enabled multi-core platform (i.e., Arm Musca-A1 board) considering the main sources of unpredictability and contention. The goal is to provide a set of empirical observations that can help elucidate the contention of shared resources on last-generation devices. There are multiple sources of unpredictability, such as shared memories, buses, caches, among others. In the study performed, the focus was to foreseen potential candidates of contention hardware sources present in Arm's Musca-A1 platform. In a closer look, and supported by the detailed architecture exposed in Section 3.1 and Figure 3.1, the memory infrastructure is supported by buses with concurrent paths between the two cores. The cores are connected to the main bus matrix, a multi-layer AHB5 interconnect, allowing parallel access paths between the multiple masters and slaves in the system. A set of four 32KB internal SRAMs (iSRAMs) elements are also accessible via the main bus. Therefore, each one of the cores can access them concurrently without expected contention. Three of the iSRAMs (0, 1, 2,) are tightly coupled (TCM) to the CPU0 while the iSRAM3 is a TCM of CPU1. Moreover, in the SSE-200, the main bus connects two slave AHB2APB bus bridges which allow access to system control registers and peripherals. Finally, in the Musca-A1 board, two expansion ports extend the SSE-200 AHB bus matrix. One of the ports is connected to a set of APB slaves encompassing I/O functionality, while the other connects two memory elements used only for code storage, i.e., code expansion. Therefore, even if each one of the elements is accessed through distinct controllers, they are still sharing the same bus, which prevents full concurrency, i.e., when it is assigned different memories of the code expansion to CPU0 and CPU1, it is expected contention.

3.2.1 Baseline Results

As previously mentioned, Arm's Musca-A1 memory subsystem is potentially a source of contention, with several interconnects and buses being shared between the two cores. This subsection will expose

the results collected of the Coremark benchmark running natively on each Musca-A1 cores for different memory layouts.

Accordingly to the guidelines provided by Arm [90] and Coremark [89], the benchmark must execute for at least 10s. Therefore, multiple runs of different iterations must be collected to verify the convergence of results (i.e., iterations per second).

The goal of the baseline experiments is to benchmark individually each Musca-A1 core, while assuming that there is contention on the system shared memory resources. Therefore, each memory layout selected are ideal, meaning that code and data sections are never placed on the same memory bank being tested. The performed experiments are structured according to the different memory components to test, which make a total of three groups: (i) internal SRAMs (i.e., iSRAM0, iSRAM1, iSRAM2, and iSRAM3), which are connected to the main bus (i.e., AHB6 Matrix) as dedicated slave components; (ii) external code-based SRAMs (i.e., eSRAM0, eSRAM1, eSRAM2, and eSRAM3), which share a single bus path; (iii) and external flash code (i.e., the QSPI Flash), also with a single path. Following this memory infrastructure, three main experiments were drawn and performed per cache state (i.e., enabled or disabled). These memory layouts are the following:

1) In the first experiment, the code is placed on the external SRAMs. The Coremark benchmark is run individually, i.e., one at a time, in each core of Musca-A1 to avoid any source of contention on the access to the eSRAMs. Results are collected for each core, and a baseline is computed.

2) In the second experiment, the code is mapped to the Flash memory. The number of iterations per second is expected to be substantially lower than the other two experiments when the instruction cache is disabled.

3) In the third experiment, the internal SRAMs are benchmarked. The Coremark code is placed on each of the four memory banks composing the iSRAMs. The test is also run with the instruction cache enabled and disabled on each core of the system.

Experiment #1 (external SRAMs) Table 3.1 depicts the Coremark results for each core of Musca-A1 when running code from the eSRAMs. The data segment is placed in each core TCM bank, which for CPU0 is the iSRAM bank one (i.e., iSRAM1), and for CPU1 is bank three (i.e., iSRAM3) - the goal is to benchmark each core under similiar conditions. The compiler is configured with optimization flag -O3 and the platform is set up with both CPUs running at 50 Mhz frequency.

			Cache					
			Enabled			Disabled		
Code	Data	Core	Iterations	Time (s)	Iteration/s	Iterations	Time (s)	Iteration/s
eSRAM0	iSRAM1 (TCM CPU0)	CPU0	1500	10.72	139.93	1500	10.67	140.58
			2000	14.29	139.96	2000	14.23	140.55
			4000	28.59	139.91	4000	28.47	140.50
			6000	42.89	139.89	6000	42.70	140.52
			8000	57.18	139.91	8000	56.94	140.50
			10000	71.48	139.90	10000	71.18	140.49
eSRAM3	iSRAM3 (TCM CPU1)	CPU1	1500	11.17	134.29	1500	38.95	38.51
			2000	14.89	134.32	2000	51.93	38.51
			4000	29.79	134.27	4000	103.87	38.51
			6000	44.69	134.26	6000	155.81	38.51
			8000	59.59	134.25	8000	207.74	38.51
			10000	74.49	134.25	10000	259.68	38.51

Table 3.1: eSRAM baseline results for CPU0 and CPU1.

As mentioned previously, the Coremark requires multiple runs of different iterations to verify the correction of tests. If the number of iterations per second is consistent, the implementation of the benchmark is valid. Therefore, in this first experiment, the benchmark is run with different iterations for each core, from 1500 to 10000 iterations. In the following experiments, only the memory layout was modified; therefore, the number of iterations is fixed since Coremark implementation's correctness is assured in this first experiment. Moreover, the decrease in iterations/s in the CPU1's case is due to the underlying architectural asymmetries of Musca-A1 design (i.e., the CoreLink SSE-200 subsystem) [11].

Experiment #2 (QSPI Flash) Table 3.2 depicts the Coremark results for each core of Musca-A1 when running code from the QSPI Flash. The data segment is placed in each core TCM bank, which for CPU0 is the iSRAM bank one (i.e., iSRAM1), and for CPU1 is bank three (i.e., iSRAM3) - the goal is to benchmark each core under similiar conditions. The compiler is configured with optimization flag -O3 and the platform is set up with both CPUs running at 50 Mhz frequency.

			Cache					
			Enabled			Disabled		
Code	Data	Core	Iterations	Time (s)	Iteration/s	Iterations	Time (s)	Iteration/s
Flash	iSRAM1 (TCM CPU0)	CPU0	1500	16.43	91.30	100	26.59	3.76
	iSRAM3 (TCM CPU1)	CPU1		16.32	91.91		26.59	3.76

Table 3.2: QSPI Flash baseline results for CPU0 and CPU1.

The results of this experiment show that the instruction cache, as expected, substantially reduces in each of the cores, the total time required to complete 15x more iterations of coremark when using the same memory layout. Moreover, both cores' QSPI Flash results are almost the same when the cache is enabled or disabled.

Experiment #3 (internal SRAMs) Table 3.3 depicts the Coremark results for each core of Musca-A1 when running code from the iSRAMs. For CPU0, the data segment is always placed inside a TCM but without ever sharing the same memory for data and code, and for CPU1 the bank three (i.e., iSRAM3) is used whenever it is not being used for code. The compiler is configured with optimization flag -OS (to fit the code into only one iSRAM bank) and the platform is set up with both CPUs running at 50 Mhz frequency.

			Cache					
			Enabled			Disabled		
Code	Data	Core	Iterations	Time (s)	Iteration/s	Iterations	Time (s)	Iteration/s
iSRAM0	iSRAM1	CPU0	1500	19.18	78.21	1500	19.18	78.21
iSRAM1	iSRAM0			19.18	78.21		19.18	78.21
iSRAM2	iSRAM1			19.18	78.21		19.18	78.21
iSRAM3				59.95	25.02		59.95	25.02
iSRAM0	iSRAM3	CPU1	1500	55.62	26.97	1500	55.62	26.97
iSRAM1				55.62	26.97		55.62	26.97
iSRAM2				55.62	26.97		55.62	26.97
iSRAM3	iSRAM2			25.49	58.85		25.49	58.85

Table 3.3: iSRAMs baseline results for CPU0 and CPU1.

Accordingly to the literature [12] and, as the results show, for both cores, the instruction cache has no impact when it is used the iSRAMs to store the code. Therefore, the results are the same. For example, the CPU0 takes 19.18s with cache enabled and disabled to complete 1500 iterations of coremark when the code is placed in the iSRAM0 and data in the iSRAM1. Moreover, when is used an iSRAM that is not a TCM to the core, the time taken to complete the same 1500 iterations increases. For instance, CPU1 takes 55.62s to complete 1500 iterations when the code is placed in a memory that is not a TCM, but it only takes 25.49s to complete the same 1500 iterations when CPU1 TCM is used.

3.2.2 Contention Results

This subsection exposes the Table 3.3 results achieved after re-running the previous three experiments (and and extra one) with both cores running concurrently. The aim is to stress the memory infrastructure and verify which memory candidates are a source of contention on the system. Hence, from all of the results obtained, the focus is directed to the amount of time that each core - now running simultaneously - takes to complete the same number of iterations. If the results show variability related to the baseline, the memory infrastructure of Musca-A1 has effectively contention points introducing unpredictability on the system.

Experiment #1 (external SRAMs) Table 3.4 depicts the Coremark results obtained for each core which are running concurrently and fetching code from the eSRAMs. The memory layout is the same as in the baseline experiment. Therefore, the data segment is placed in each core TCM bank, which for CPU0 is the iSRAM bank one (i.e., iSRAM1), and for CPU1 is bank three (i.e., iSRAM3). The compiler is configured with optimization flag -O3 and the platform is set up with both CPUs running at 50 Mhz frequency.

				Cache			
				Disabled		Enabled	
Code	Data	Core	Iterations	Time (s)	Iteration/s	Time (s)	Iteration/s
eSRAM0	iSRAM1 (TCM CPU0)	CPU0	1500	18.80	79.79	10.75	139.53
eSRAM3	iSRAM3 (TCM CPU1)	CPU1		44.52	32.07	11.18	134.17

Table 3.4: eSRAM code contention results.

In this experiment, the Coremark is configured to run 1500 iterations, since its correct implementation was verified in subsection 3.2.1. As it can been seen in Table 3.4, the benchmarked time of each core when the cache are disabled as increased substantially. The results show a performance degradation of -43% for CPU0 (18.80s) and around -14% for CPU1 (45.58s). This considerable interference imposed in the system while both cores fetch instructions from the external SRAMs results in a lack of determinism and timing predictability. Moreover, regarding the cache-enabled results, the performance degradation is almost residual; for CPU0, the degradation is around -0.3% and for CPU1 is around -0.09%. Such values can only express a good code locality of the Coremark application, meaning that the benchmarks access the same set of memory locations from the already cached code. Since each core has a dedicated instruction cache, the interference caused by the external SRAM is almost null in the presence of good code locality.

Experiment #2 (QSPI Flash) Table 3.5 depicts the Coremark results obtained for each core which are running concurrently and fetching code from the QSPI Flash. The memory layout is the same as in the baseline experiment. Therefore, the data segment is placed in each core TCM bank, which for CPU0 is the iSRAM bank one (i.e., iSRAM1), and for CPU1 is bank three (i.e., iSRAM3). The compiler is configured with optimization flag -O3 and the platform is set up with both CPUs running at 50 Mhz frequency.

			Cache					
			Disabled			Enabled		
Code	Data	Core	Iterations	Time (s)	Iteration/s	Iterations	Time (s)	Iteration/s
Flash	iSRAM1 (TCM CPU0)	CPU0	100	94.81	1.05	1500	19.62	76.45
	iSRAM3 (TCM CPU1)	CPU1		–	–		19.52	76.84

Table 3.5: QSPI Flash code contention results.

As it can be seen in Table 3.5, the benchmarked time of CPU0 when the cache is disabled has increase substantially. The results show a performance degradation of -71% for CPU0 (94.81s). This considerable interference imposed in the system while both cores fetch instructions from the QSPI Flash results in a lack of determinism and timing predictability. Moreover, regarding the cache-enabled results, the performance degradation is significantly reduced; for CPU0, the degradation is around -16% and for CPU1 is also -16%. Such performance degradation is a derivation of good code locality and the existence of individual instruction caches for each one of the cores.

Experiment #3 (Internal SRAMs) Table 3.6 depicts the Coremark results obtained for each core which are running concurrently and fetching code from the iSRAM. The memory layout is the same as in the baseline experiment. Therefore, for CPU0, the data segment is always placed inside a TCM but without ever sharing the same memory for data and code, and for CPU1 the bank three (i.e., iSRAM3) is used whenever it is not being used for code. The compiler is configured with optimization flag -Os and the platform is set up with both CPUs running at 50 Mhz frequency.

					CPU0		CPU1	
CPU0		CPU1			Cache Disabled			
code	data	code	data	Iterations	Time(s)	Iteration/s	Time(s)	Iteration/s
iSRAM0	iSRAM1	iSRAM2	iSRAM3	1500	19.18	78.21	55.62	26.97
iSRAM1	iSRAM2	iSRAM3	iSRAM0		19.18	78.21	25.49	58.85
iSRAM2	iSRAM3	iSRAM0	iSRAM1		19.18	78.21	55.62	26.97
iSRAM3	iSRAM0	iSRAM1	iSRAM2		59.95	25.02	55.62	26.97
iSRAM0	iSRAM1	iSRAM3	iSRAM1		19.95	75.19	27.19	55.17
iSRAM0	iSRAM2	iSRAM3	iSRAM2		19.95	75.19	27.19	55.17

Table 3.6: iSRAM data and code contention results.

Table 3.6 shows that the benchmarked time of each core when the cache are disabled or disabled is the same. For instance, it takes 19.18s for CPU0 to complete 1500 iterations of Coremark when CPU1 is disabled and enabled and using different iSRAMs. The results show that there is no performance degradation as long as none of iSRAMs is shared for data or code. Moreover, regarding the last two lines of the table it shows that when the same iSRAM is shared for data, there is a slight performance degradation; for CPU0, the degradation is -3.85% and for CPU1 is -6.25%. Another purpose of this experiment was to show that the bus access matrix introduces no type of contention when cores are running concurrently.

Experiment #4 (Hybrid Layouts) Table 3.7 depicts the Coremark results obtained for each core which are running concurrently and fetching code from different memories. The memory layout follow the same baseline model. Therefore, for CPU0, the data segment is always placed inside a TCM (i.e., iSRAM1), and for CPU1 is used the iSRAM third bank (i.e., iSRAM3). In the first experiment, the code of CPU0 is placed in the QSPI Flash, with compiler optimization flags of -O3 and, in the last two experiments is used the iSRAM0 with the compiler optimization of -Os. The code of the CPU1 is placed in the eSRAM3 for the first two benchmarks and in the last is placed in the Flash memory with the compiler optimization flags of -O3.

				CPU0			CPU1		
CPU0		CPU1		Cache Disabled					
code	data	code	data	Iterations	Time(s)	Iteration/s	Iterations	Time(s)	Iteration/s
Flash	iSRAM1	eSRAM3	iSRAM3	100	53.21	1.88	1500	–	–
iSRAM0	iSRAM1	eSRAM3	iSRAM3	1500	19.18	78.21		38.95s	38.51
iSRAM0	iSRAM1	Flash	iSRAM3		19.18	78.21	100	26.59s	3.76

Table 3.7: Hybrid contention results.

The results of this experiment show that the core has no performance degradation when it uses different iSRAMs for code and data even if the CPU1 is using one of the expansion code memories. For example, CPU0 takes 19.18s to complete 1500 iterations for coremark when running solo or with CPU1 using the QSPI Flash or eSRAM for code. On the other hand, when the different CPUs use different memories from the code expansion (i.e., QSPI Flash and eSRAM3), there is a performance degradation of around -50% for CPU0. Therefore, the premise that both cores can not access different expansion code memories concurrently is confirmed.

3.2.3 Discussion

The proposed benchmarks showed that when there is good code locality, the instruction cache beyond increasing the system performance in most situations also increases the system determinism. Moreover, the benchmarks also confirmed that the instruction cache has no sort of impact on the iSRAMs.

When used different iSRAMs for each core, the overall result proves the premise that there is no contention when it is concurrently access to the main bus matrix by both cores. Moreover, each one of the cores has their code placed inside of different eSRAMs, a significant increase in the benchmarking timings leading to the decrease of the system's predictability. Even when both cores use different code expansion memories (e.g., CPU0 eSRAM0 and CPU1 QSPI Flash), the results show that there is the contention at the bus level due to the impossibility of multiple masters control the same access bus concurrently. Therefore code contention can come from the simultaneous use of expansion code memories by both cores.

External to the results demonstrated in the preliminary results, it was also tested the impact of the use of the instruction cache in one of the cores only. The results showed that when one of the cores has the

instruction cache enabled and has its code in QSPI Flash memory, while the other cores use the eSRAM memory or the QSPI Flash memory, there is still jitter introduced into the system behavior.

In terms of data contention, since the iSRAMs are the only memories that can be used for their placement, it only occurs when the same iSRAM is shared by both cores.

To conclude, when testing the mechanism at the code level, three memory layout can be used:

- both cores with cache disable running from the eSRAMs;
- one core with the cache enabled running from the Flash memory and the other core using the eSRAM or QSPI Flash memory;
- both cores running with cache enabled from the Flash memory.

Moreover, there is only one possible layout for testing data contention that counts in sharing the same iSRAM for both cores data and their code placed in different iSRAMs.

3.3 KIR: System Architecture

In this section, the design of the proposed mechanism, i.e., KIR, is exposed. KIR aims at reducing the contention over shared resources on low-end multi-core platforms. Based on literature review and state-of-the-art solutions, KIR proposes a memory bandwidth reservation solution, focusing on controlling memory throughput to increase the system's predictability at the expense of performance (a necessary tradeoff to improve the system's determinism).

Since the hardware design of each platform has a high impact on which resources introduce contention points on the system, KIR was designed to leverage architecture-specific mechanisms to scale the solution to other Armv8-M devices. Therefore, KIR sits on top of Armv8-M-based devices and explores underlying modules feature on the CoreSight debug subsystem to implement the reservation mechanism.

According to the literature [12, 80, 3], the Arm Musca-A1 board does not implements all of the debug and tracing hardware described in state of the art (section 2.1.1.5). Therefore, the design of KIR is restricted. From the assessment made, the elements that stand out are the DWT, the ETM, the ITM for tracing and the TPIU as the solo tracing sink; thereby, all of the generated trace data can only be exported to an external tool. Therefore, it is impractibale the use of non-invasive type of debugging, meaning that the only available technique is the invasive type of debugging, i.e., Monitor Debug.

KIR must support the auto-regulation of the system during its execution without outside intervinience; thereby to collect real-time CPU consumptions, the DWT unit is used. The CPU consumptions are break-down into two parts known as data and code. Data represents the heap, stack, and all of the global variables, i.e., *.bss .data* sections, and code represents all of the instructions, read-only variables and VTOR, i.e., *.text* section. Figure 3.2 depicts the layers of KIR and each one of the components that composed the mechanism.

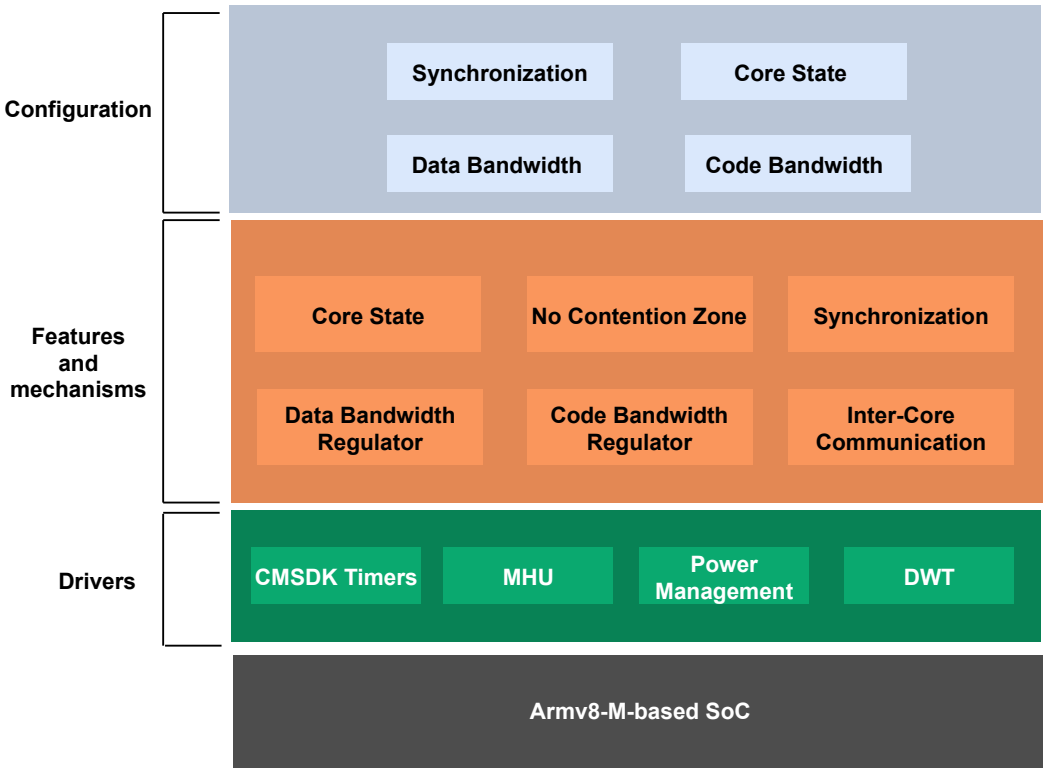


Figure 3.2: KIR Architecture.

Code Bandwidth regulator: This regulator is responsible for containing the number of instructions that a selected core can perform in a certain period of time. Therefore, KIR controls the core bandwidth depending on the maximum number of instructions that the core can execute. The regulator exploits the DWT counters to monitor and calculate the number of instructions executed by the core. The relative instruction count is governed by the equation (1) [91].

$$\text{Relative instruction count} = \text{CYCCNT} - \text{CPICNT} - \text{EXCCNT} - \text{SLEEPCNT} - \text{LSUCNT} + \text{FOLDCNT} \quad (1)$$

When comparing the DWT with other monitoring units of high-level platforms, such as the PMU on Cortex-A processors or the PMC on Intel-based devices, the DWT does not allow to interrupt the system once an

instruction count value is reached. Therefore, the code regulator needs to use a counter/timer to cause synchronous exceptions for the collection of this valuable information (e.g., DWT CYCNT function match with comparator 0). In each interrupt, the value of instruction count must be updated by adding the new relative instruction count. Therefore, the total of instructions at given time is calculated by the equation (2).

$$\text{Instruction count} \leftarrow \text{Instruction Count} + \text{Relative Instruction Count} \tag{2}$$

Data Bandwidth regulator: This module watches a memory address or range of memory addresses used for data. Watching memory requires two DWT comparators: (i) the first one must be set with the starting data address; (ii) and the second one as data address limit. With the monitor debug exception enabled, for each write or read, an exception is thrown [3], which allows the regulator to increment a software counter that keeps track of the number of data accesses. Therefore, at a given time of the exception, the data memory access is given by the equation (3):

$$\text{Data count} \leftarrow \text{Data Count} + 1 \tag{3}$$

To prevent a core from executing any more instructions, KIR will set the specific core to a sleep state. Therefore, when a threshold of instructions or a number data memory accesses is reached, the KIR will change the core running state to a sleep state. Each core has limited data and code bandwidth, set by the user, that it can consume in a certain period of time. At the end of each period, the bandwidth budgets are replenished. As shown in Figure 3.3, for a given period, the core can only perform a certain maximum of instructions/data accesses. Both KIR bandwidth regulators are independent of each other, but both depend on the execution window time given by the end user.

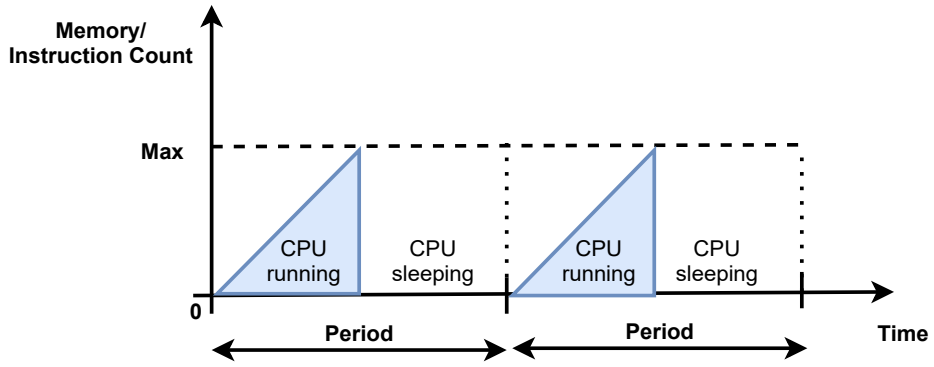


Figure 3.3: Example of KIR execution window.

Regulating the bandwidth of the core can lead to periods of execution where the cores are not overlapping and, therefore, not competing for the control of the same resources, which are typically the buses used to address the code expansion memories in this platform. Moreover, each of the core has its own KIR; thereby, each of the cores has its own regulators and period. Since CPU0 boots first, then CPU1, its period will start first, leading to decentralized execution windows, with the T time size, as shown in Figure 3.4. Therefore, allied to the regulators are two features named synchronization and core state. In conjunction, allow for the periods to start at the same time and, if desired, invert the execution time of the core, i.e., *sleep first, execute later*.

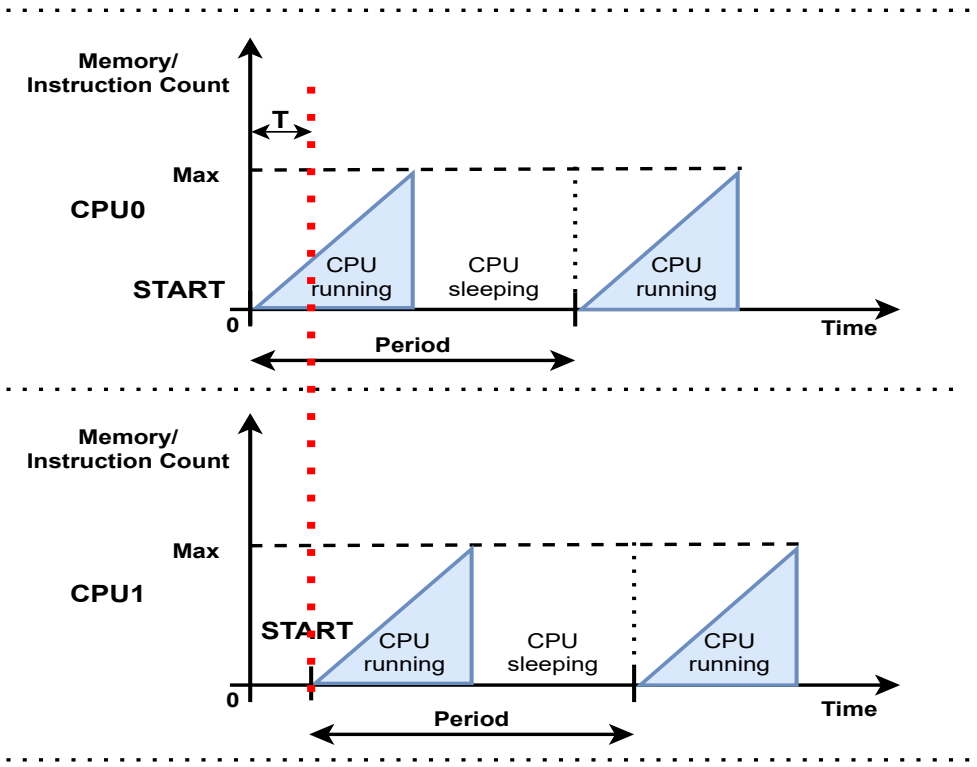


Figure 3.4: Example KIR not synchronized between cores.

Synchronization feature: Synchronization does not consist of both cores starting their execution simultaneously but instead enabling the mechanism at the same time. Thus, synchronization can be achieved by starting both periods of the different cores at the same time. Therefore, their execution windows will be overlapped all the time, i.e., if both cores have the same bandwidth, meaning that both cores will run and sleep at the same time, and, therefore, the contention maximized. This functionality can only be carried out by exchanging messages between both cores during run-time; thereby it is required the message handling unit existent in the SSE-200 [80]. The messages are transmitted via limited channels implemented in the MHU, and when a message is received, an interrupt is triggered (if enabled). In the handler, after reading the message, the channel must be cleaned. Moreover, the messages sent have simple payloads, with a value associated with them, and are statically defined (see Table 3.8).

Message	**Value Associated**
MessageClear	0
SyncRequest	1
SyncConfirm	2
SleepRequest	3
SleepConfirm	4
WakeUpRequest	5
WakeUpConfirm	6

Table 3.8: KIR inter-core messages.

There are two points where the cores must synchronize (i) during boot time and ; (ii) every *n* number of periods. Synchronization over execution is required due to the competition for the control of the shared resources (e.g., memory controller, shared bus) and, therefore, delays that will eventually cause the periods to be out of sync. Moreover, since CPU0 is the first to boot, it is defined as the master and CPU1 as the slave. In other words, any synchronization to be done is controlled by CPU0.

During boot, CPU0 must initialize the regulators if set, and if the synchronization of the system is configured, a *SyncRequest* must be requested to CPU1, and then wait for a *SyncConfirm* response. On the other hand, CPU1, after initializing KIR, must wait for a *SyncRequest* and, after received, must clean its counters and send a *SyncConfirm* to CPU0, and only after enabled them.

To define the synchronization points on every *n* periods, an extra timing source must be used (e.g., Timer 1). For each tick of the timer, the CPU0 resets its counters, sends a *SyncRequest* to CPU1, and

then enters a sleep state until it receives a response. CPU1, when receives a *SyncRequest*, cleans its counters, sends a *SyncConfirm* and only after enables the counters again. Using a timer to enable the synchronization of cores through message exchanging removes the need to synchronize the cores by a pulling mechanism. Figure 3.5 depicts a final system with both cores synchronized while using the KIR mechanism.

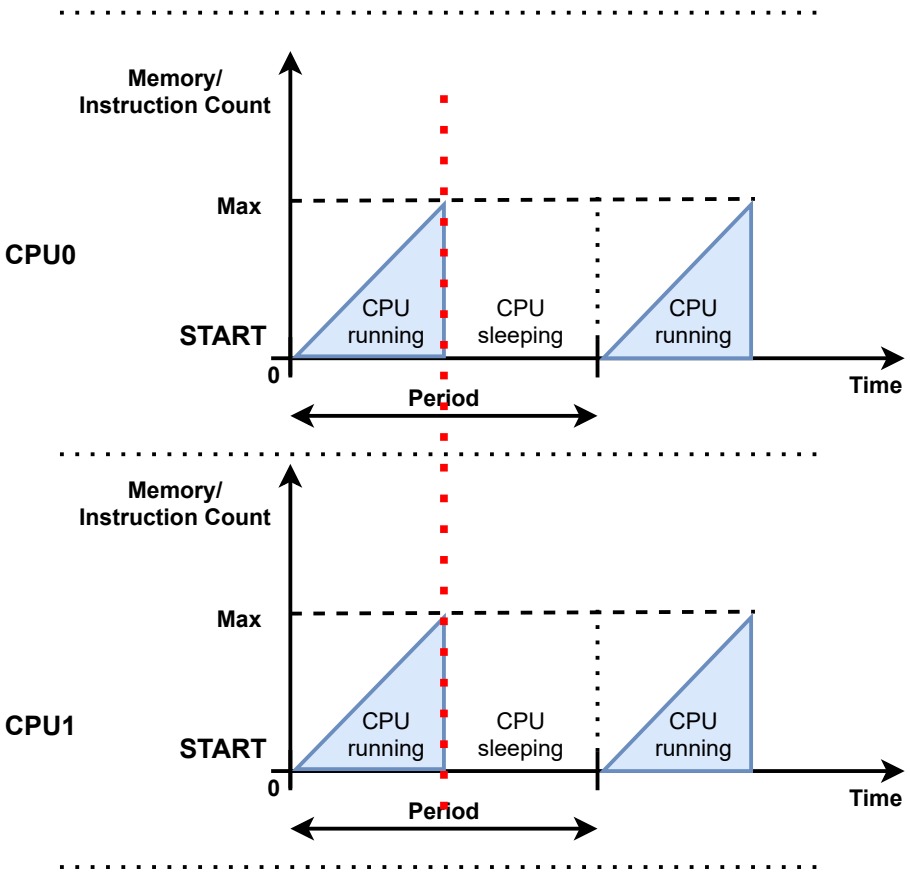


Figure 3.5: Example of KIR synchronized between cores.

Core State feature: The KIR architecture features another mechanism called Core State. This module aims to invert the execution of the core's running window after synchronization has been done. This is achieved by forcing a core to stay in a sleep state for a given T period. When enabled and combined with the synchronization mechanism, a known and controlled execution delay (the T time) can be created between both cores mechanism. Thus, both cores can execute simultaneously without any contention concerning memory accesses done by a shared bus, as shown in the Figure 3.6.

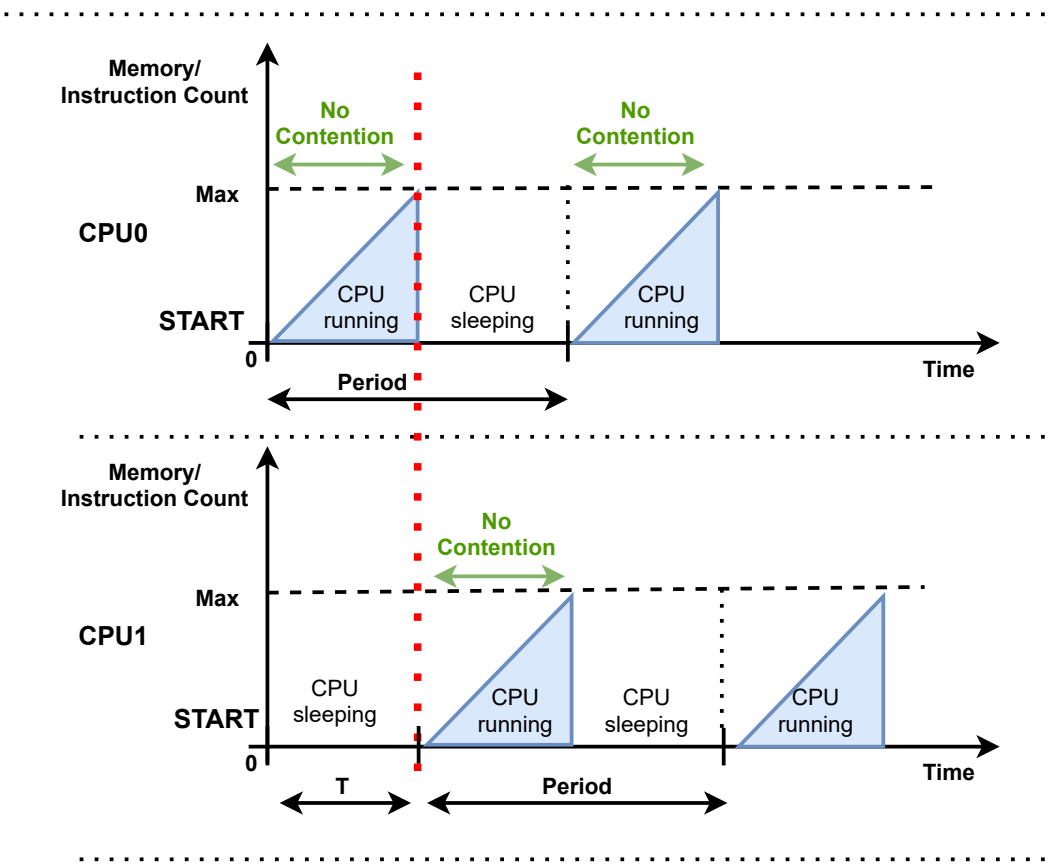


Figure 3.6: Example of KIR core state with cores synchronized and in opposed execution phases.

No Contention Zone mechanism: In addition, KIR includes a mechanism named *No Contention Zone*. The principle behind the *No Contention Zone* mechanism is to allow one core to ask the other to enter a sleep state while executing critical code. If the one of the cores is in a sleep state, there is no contention.

The mechanism consists of using the inter-communication subsystem to send different messages with the purpose of one core ask the other to enter a sleep state. Hence, the core using this mechanism as full-free execution without being bound by the regulator or sharing any resources. Such functionality is ideal for executing high-predictable critical code since it avoids contention overhead introduced by shared resources. Once it finishes running, it must send another request to wake up the sleeping core and re-enable all of the statically configured KIR mechanisms.

Four messages can be sent, where for each request, there is a response associated with it. When it is desired to start a *No Contention Zone*, the core sends a *SleepRequest* and waits until it receives a *SleepConfirm*. Once it is received a response, the KIR mechanism is disabled, and the core runs freely.

On the other hand, when a core receives a *SleepRequest*, it disables the KIR mechanism, sends a *SleepConfirm*, and then enters a sleep state. After finishing the execution, a *WakeUpRequest* is sent, and then it must wait until a *WakeUpConfirm* is received. Once it is received, the core re-enables KIR and its features. Moreover, when a core receives a *WakeUpRequest*, it re-enables KIR and sends a *WakeUpConfirm*. Figure 3.7 shows an example of the use of the *No Contention Zone* mechanism.

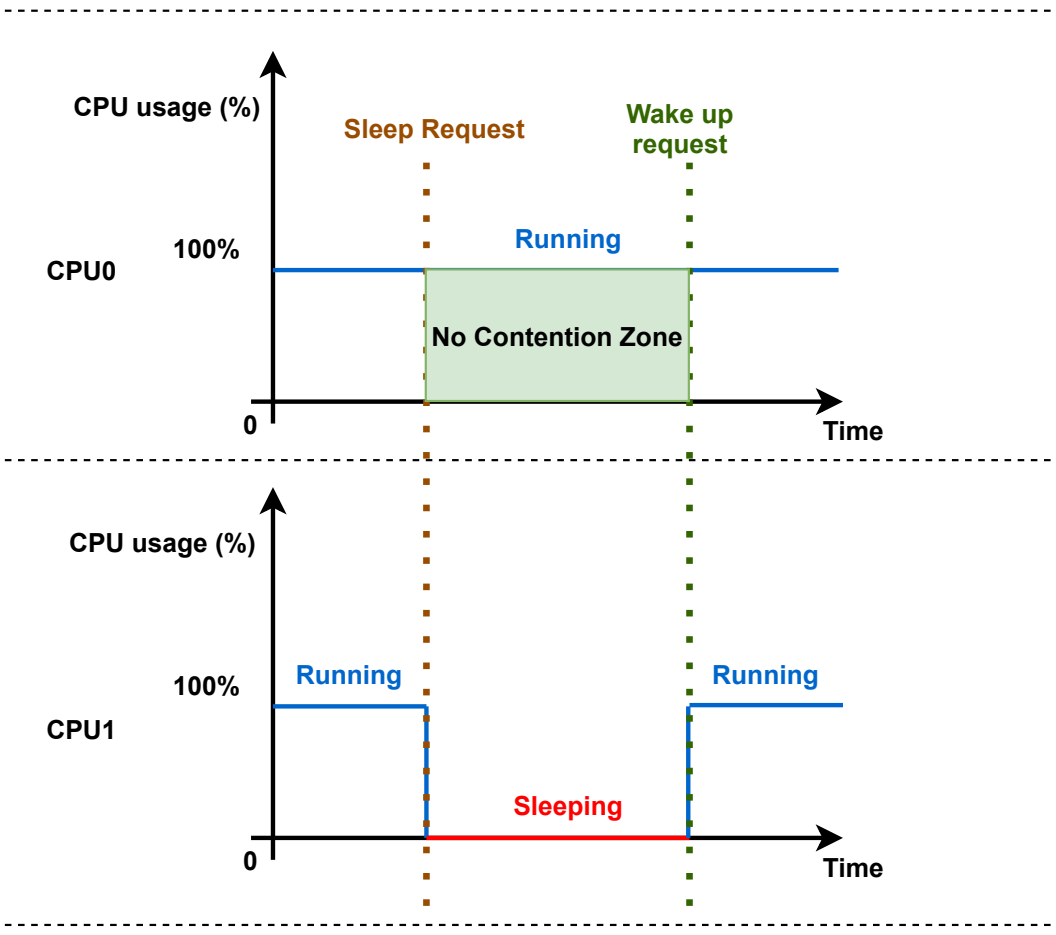


Figure 3.7: KIR No Contention Zone.

Additionally, suppose the *No Contention Zone* is used, and the *Synchronization* and *Core State* feature are enabled; this can result in another automatic period where the core runs, although limited, without contention. For example, if CPU1 has the *Core State* feature enabled and CPU0 uses a No Contention Zone, when it ends, it will automatically take another T period without any contention. Figure 3.8 frames how the combination of all of the features would lead to.

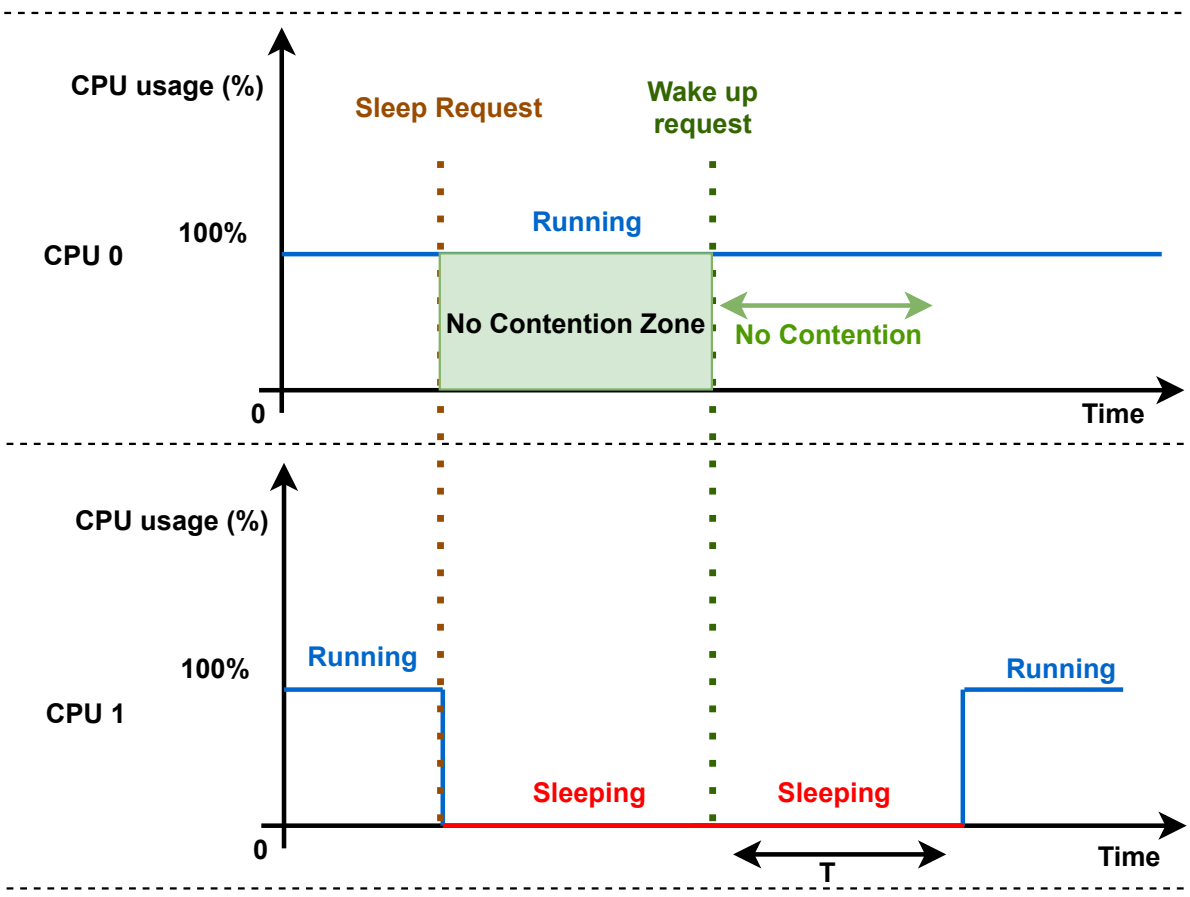


Figure 3.8: KIR Core State with cores synchronized and in opposed execution phases when no contention zone is used.

In sum, the KIR mechanism reduces the cores competition over bus accesses by alternating the execution between running and sleep states. After the core is reset, if the core state feature is enabled, its state will be equal to *sleep* (case B) in Figure 3.9). Otherwise it stays in a *running* state (case A)). During execution, transitions from running to a state of sleep are originated by multiple events (case E)) such as : (i) maximum memory accesses reached; (ii) maximum instruction executed reached; (iii) sleep request from other core; (iv) synchronization event request; (v) sleep time start. The core maintains the running state while it does not exhaust the bandwidth budget or if no *SleepRequest* is made (case C)). Moreover, transitions from sleeping to a running state (case F)) can be caused by: (i) period time elapsed; (ii) wake up request; (iii) synchronization event response; (iv) and sleep time elapsed. The core sleeps until a new period starts, considering the core state is disabled, and if no *WakeUpRequest* is realized (case D)).

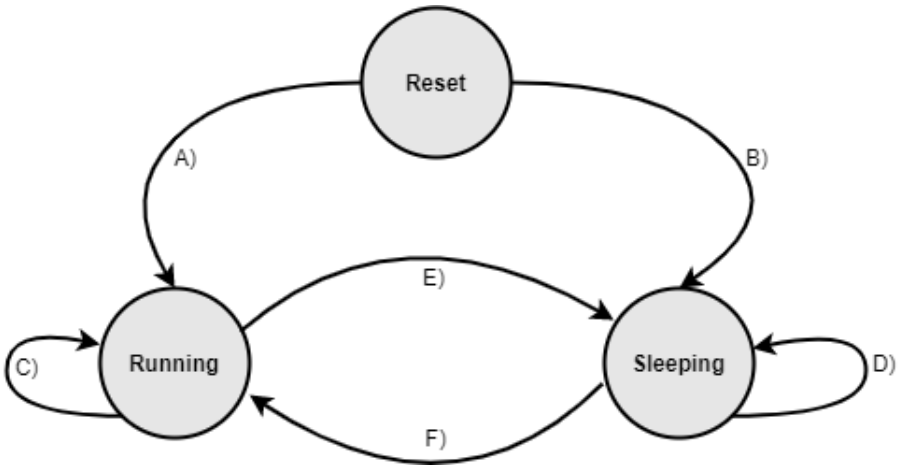


Figure 3.9: CPU possible states.

3.3.1 Integrating KIR in uTango

As previously explained, uTango, is an open-source in-house developed TEE that is going to be used to test KIR in a more realistic scenario. Therefore, in order to integrate KIR, it is required some changes.

As mention in Section 2.1.2, the uTango TEE supports Armv8-M-based platforms, running in a single-core configuration. To integrate KIR into uTango with the goal of improving the TEE's predictability, uTango needs to support a multi-core configuration. Such support needs to preserve uTango's coding structure and style (e.g., avoid code files redundancy, code separation by each core).

Moreover, the proposed mechanism, KIR, has different features and mechanisms, namely regulating the code and data bandwidth, the synchronization between cores, and inverting the core's execution phase by using the *Core State* feature. Until now, all of the features targeted the entire core since it was only focused on a ratio of one application per-core. On the opposite, uTango can have more than one application running per-core. Hence, some changes are required to be made to the mechanism for its integration in the TEE.

Single-core to Multi-core: On Arm's Musca-A1 platform, the CPU0 is defined as the main processor, meaning it is the first to boot, and it starts the execution of CPU1. The CPU0 achieves this by configuring the CPU1 VTOR register and then waking up the processor by cleaning the CPUWAIT register of the SCR block [12]. To ease the porting work of uTango to a multi-core configuration, the Asymmetric Multi-Processing (AMP) architecture was selected. An AMP configuration consists of splitting the secure kernel into two individual parts. Each one of the cores have their own kernel and associated VMs with separated address spaces. With such configuration, each core will run a version of the TEE as it can be seen in Figure 3.10.

Regarding the build and linking system, some changes must be taken: (i) the former need to be modified to build two separated binaries, while (ii) the latter needs to contemplate the memory map of the second core through a second linker script.

KIR Regulators: Instead of regulating the entire core, KIR now focus in each one of the running worlds. Therefore, each one of the worlds will have its own bandwidth limits. In the init of uTango, it is reset all of the code and data bandwidth counters and, from their one forward, the application is limited to those bandwidths. During a context switch, the budgets are replenished, and the newer application starts to run. In case the number of applications is less than one, all of the values of the counters are maintained and, the application proceeds with its execution.

KIR Synchronization Feature and uTango Scheduler: For the integration of the synchronization feature, it was opted to use the scheduling points of uTango to force a synchronization if enabled. This way, it is reduced the number of resources used and, at the same time, the complexity of the TEE static configurations. To increase the flexibility of the synchronization feature configuration, it was added an extra counter that increments at each scheduling point. With this extra counter, it is possible to increase the period of each synchronization point without the need of having an higher uTango tick. For example, if each context switch happens every 10ms, it is possible to set synchronization points for every 500ms, i.e., only after 50 scheduling points a synchronization occurs. Moreover, the synchronization feature continues to target the entire core instead of a single world.

KIR Core State Feature: Since the Core State feature is tightly coupled with the synchronization, its target also remains the core and not a single world. Moreover, when the core is in the secure world (i.e., executing uTango), when it is required to change from a running state to a sleep state, an alias to the non-secure must be used. Therefore, when the core restores its execution on the non-secure side, it will enter a sleep state.

KIR No Contention Zone: The mechanism maintains its functionalities but with the key difference of how it is accessed due to security state of the application running (i.e., non-secure). Hereafter, it is required that the non-secure application uses an NSC entry point provided by uTango to start a No Contention Zone. Thus, from any point in the non-secure world, can be used this mechanism.

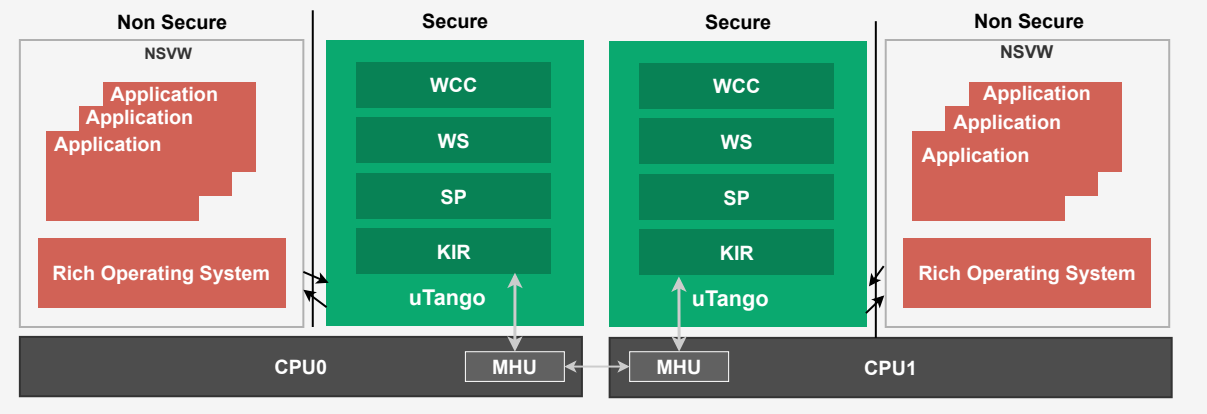


Figure 3.10: uTango TEE with KIR integrated in a AMP configuration.

3.4 Experimental Setup

This section of the chapter specifies the scenarios in which the KIR mechanism introduced in this dissertation will be tested and verified. More specifically, a set of tests will be carried out in different scenarios, designed to collect data necessary for validation of the mechanism. Both of the designed scenarios focus on the system's code and data bandwidth. Coremark [89] is used as the benchmark to collect metric data about the developed mechanism. The Coremark suite requires the support of a timer unit to calculate the timing operations and a UART to transmit the results. Hence, throughout the tests, different memory layouts are used to achieve maximum contention at the code or data level, in a similar approach as in subsection 3.2. For example, to validate that the code bandwidth subsystem is acting as expected, the data of both cores must be stored in the corresponding TCM, where no contention is expected, and the code in the shared code memories such as the external SRAMS and the QSPI Flash Memory.

The experimental setup will test KIR running and acting side-by-side with the Coremark suite. The benchmarks are structured to verify each of the main three components of KIR: (i) the code bandwidth regulator; (ii) the data bandwidth regulator; (iii) and the No Contention Zone. Moreover, each synchronous exception's tick interval and the period of execution are maintained throughout all tests.

3.4.1 Testing the Code Bandwidth Regulator

For testing the code bandwidth regulator, it is essential to consider multiple factors such as the location of the code and data of Coremark benchmark, the KIR mechanism and the cache status. Taking into

account the preliminary results exposed in the subsection 3.2, the eSRAMs or the QSPI flash memories introduce contention points on the system when shared between the cores. With the goal of introducing contention interference on the system, such memories are selected for code placement. Since this particular test is focused on the code bandwidth regulator, the data section must be placed in contention-free memories, i.e., the different iSRAMs, to avoid undesired contention derived from data accesses. The instruction caches are disabled during the test to create the worst-case execution time. Based on the metric results collected and presented in subsection 3.2, the eSRAMs are selected to place the code of both instances of Coremark. To the CPU0 is attributed the eSRAM0, and to the CPU1 the eSRAM3. Regarding the data sections, they are placed in the iSRAMs, i.e., iSRAM1 for CPU0 and iSRAM3 for CPU1.

Considering the KIR ability to regulate the code memory bandwidth, new metric results must be collected for both cores, taking into account the maximum number of instructions the core can execute in a certain period. By selecting the maximum number of instructions the core can perform in a predetermined period, it is possible to run multiple benchmarks with the same layout of memory while changing the system's bandwidth. Thus, it is required two steps to gather all the metrics needed: (i) count the maximum number of instructions the core can perform in a period of time; (ii) test how much time it takes for the core to complete the benchmark when limited to different bandwidths. Therefore, to know the maximum number of instructions the core can execute in a certain period, it is necessary to define the tick interval of the synchronous exception and the period time. For each exception, the total instruction count must be incremented until a new period begins. Moreover, another test of the Coremark must be added in order to select the size of each tick of the synchronous exception so that the tick is not too small, and the interrupt overhead turns the mechanism unviable; and not too large to the point that the measuring of the instruction count becomes unviable due to the overflow of the DWT 8-bit counters. Throughout the benchmarks, each core must complete multiple solo runs with KIR code bandwidth enabled and limited at three different limits, x, y, and z, where x is the maximum bandwidth available and $z < y < x$. With the metric results when each core is regulated, newer benchmarks can be projected, taking into account the simultaneous execution of both cores. The contention results will then determine if KIR impacts the system's determinism.

3.4.2 Testing the Data Bandwidth Regulator

Testing the data bandwidth regulator needs a memory layout that forces contention points on only data accesses. As verified in subsection 3.2, no contention occurs when the iSRAMs are used since each

bank is defined as a slave on the bus matrix.

Therefore, to force contention, it is required for both cores to have their data inside the same iSRAM bank. Moreover, the code of each core must be loaded into the iSRAMs to avoid contention during code accesses. Based on these premises, CPU0 and 1 must use iSRAM2 for their data, but CPU1 must have an offset bigger than the size of the CPU0 data memory block, so that there is no memory overlap. For code, it is used iSRAM0 for CPU0 and iSRAM3 for CPU1.

It is vital to know how invasive this method is when observing memory with the DWT comparators. Each read or write to one of the positions marked will trigger an exception. Therefore, a tradeoff between performance in favor of higher determinism is inevitable. Furthermore, the data segment watched by the regulator needs to be outside the stack section due to the unwinding operation which occurs during exception handling. If observed, it can lead to an infinite loop of exceptions since the region used to store information when an interrupt/exception is attended will be accessed again when leaving the handler, causing another exception.

The system's data bandwidth depends on what part of memory is being watched and how many times it is expected to be used. For example, it can monitor an array of data used on each iteration of the benchmark or the entire heap section. Additionally, just as in the code bandwidth tests, it is required to collect how many data access (j) the core can perform in a certain time period. Knowing each core maximum's data bandwidth, multiple runs with lower bandwidth levels are completed. For example, a run with x, y, and z levels of data bandwidth where $x < y < z < j$. The results collected from each core are later used as comparison metrics when both cores share the same internal SRAM for data.

Having the metric results, multiple Coremark runs are projected to verify if the KIR data bandwidth regulator improves the system predictability. During the benchmark, CPU0 maintains the same limits of data bandwidth for every three runs while CPU1 goes over the three different bandwidth limits. This is done so that all possibilities are exhausted. In the end, the results must be compared to the baseline ones when the core is also limited.

3.4.3 Testing the No Contention Zone

To test the KIR's *No Contention Zone* mechanism, both cores must be initialized and then enabled. From there, a request is made a SleepRequest to CPU1. Once a SleepConfirmation is received, the core starts freely executing the 1500 iterations. Once CPU0 finishes all of the iterations, it must send a WakeUpRequest to CPU1 and then stay in a while loop. The sequence diagram presented in the figure

3.11 shows the sequence of events to be realized in this test. Therefore, CPU1 results are expected to be equal to the sleep time plus the time it takes to complete 1500, while the effects of contention are caused by CPU0 and the bandwidth regulation. The bandwidth limit for CPU1 can be any of the three possible values previously used (x, y, z). For CPU0 it is expected that the results will match the ones equal to the metric when running solo without the effects of the bandwidth regulation.

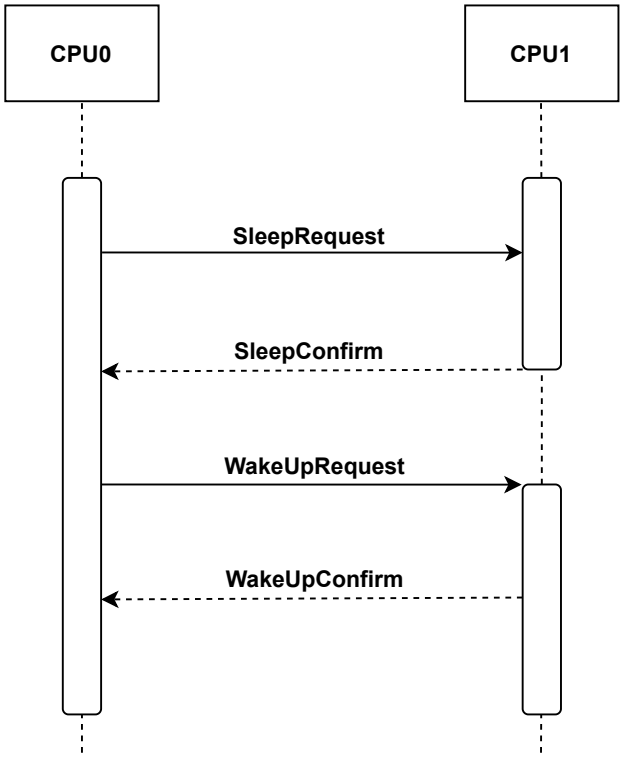


Figure 3.11: Sequence diagram of the No Contention Zone Benchmark.

Chapter 4

System Development

The previous chapter exposed KIR design and the problems that it proposes to tackle. Consequently, this chapter aims to specify the implementation of the KIR system. Using a bottom-up approach, KIR is projected to be modular and with low-dependencies so that engineering effort becomes reduced when porting is required.

Figure 3.2 shows that KIR is divided into four different layers, where: (i) the lowest one is constituted by the hardware resources such as the DWT, MHU, Timers, and Power Management facilities; (ii) the second layer is composed by the drivers with the purpose of providing a more straightforward software interface to the platform assets and abstract from the configuration registers; (iii) the third layer (i.e., middle layer) that consists of the KIR features and mechanisms' data and logic (e.g., Bandwidth Regulators, Synchronization) (iv) and in the top layer, also known as the configuration layer used to configure and abstract the end user from the logic of the lower layers.

Section 4.1 exposes the work involved in the development of the multiple drivers used. It details the implementation of each software interface using a low-level language, i.e., C programming language. Section 4.2 involves explaining the middle and top layer, supported by the drivers, that compose the mechanism. It is first presented the bandwidth regulators and, then the *Synchronization* and *Core State* features. The last section is used to report the KIR integration in uTango.

4.1 KIR Drivers

This section exposes the system drivers' main implementation details to be used by the multiple mechanisms proposed in this dissertation. It provides an overview of the software interface developed for easier interaction with the platform's resources. As described in Section 3.3, the KIR mechanism uses

four different hardware modules (Figure 4.1): (i) Power Management; (ii) Message Handling Unit; (iii) Data Watchpoint and Tracing; (iv) and a Timer.

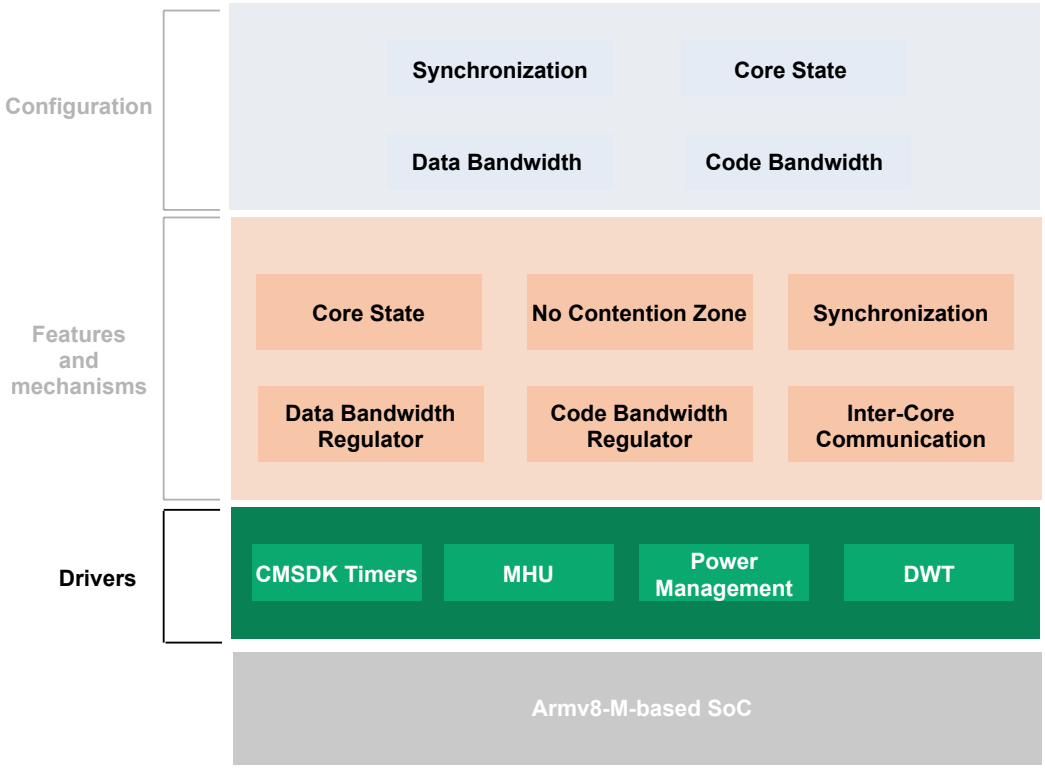


Figure 4.1: KIR Architecture: Drivers Layer.

Power Management: The drivers focus on controlling the core running state by exploring the power management facilities available in the Arm Cortex-M33. Based on the literature [51, 80, 3], the core can switch between a running and a sleep state by toggling the bit SLEEPONEXIT; thereby, when equal to 1, the CPU enters a sleep state when it transitions from Handler to Thread mode and, vice-versa. The SCR containing the SLEEPONEXIT bit is internal to each core. Therefore, the driver does not require any changes when it comes to port from single-core to multi-core.

Message Handling Unit: The Arm Musca-A1 development board implements two Message Handling units, named MHU0 and MHU1, mapped into two different memory locations. Each core has a register associated with both MHUs, that stores the received message and the state of the interrupt. The driver's primary functions are used to check the status of the selected MHU, clear the Interrupt Register associated with the unit, or read the register storing the value received. Therefore, three functions were defined: (i) *status* to read the messaged the last received message; (ii) *set* to send a message; (iii) *clear* to zeroize the interrupt flag raised when a message is received.

CMSDK Timer: The CMSDK Timer offers an interface to the timers available on the platform. According to the reference [12], the platform has two timers mapped into different memory locations. Depending on the chosen timer, the driver offers the ability to configure it and then enabled it by using a set of available functions: (i) *init* that receives as parameters the memory address of the Timer, the starting value for the counter used to set the RELOAD value, enables the interrupt, and selects the clock source (internal or external); (ii) *enable/disable* to start/stop the timer; (iii) and the *flag clearence* used to clear the interrupt flag when the counter overflows.

Data Watchpoint and Tracing: The DWT driver allows the configuration and manipulation of the unit comparator functions and their counters. To be able to configure the DWT unit, it is needed to enable tracing in the platform by setting the bit TRCENA in the Debug Exception and Monitor Control Register (DEMCR) and unlocking write accesses to the unit by writing 0xC5ACCE55UL to the DWT Lock Access Register (LAR) [3].

Each of the DWT comparators has a function register used to set the behavior of the comparator. The driver offers two main functions: (i) one to configure DWT comparator 0 to trigger an exception when the value matches the DWT CYCCNT; (ii) second used to configure two pair two comparators to watch a block of memory. The block of memory can be either code or data. The function receives the start and size of the block of memory. In this case, it was selected comparators 2 and 3. The comparator 2 is set as the base address and to create a debug event on a match and the comparator 3 as the limit address. The driver also offers a function that returns the relative instructions (see equation 1 in Section 3.3). Lastly, there are two functions named *enabled* and *disabled* the used to enable or disable DWT counters (e.g., CYCCNT, FOLDCNT, LSUCNT). The DWT counters are enabled/disabled by the DWT Control register (CONTROL). Lastly it was added a *reset* function used to reset the DWT counters.

4.2 KIR Middle and Top Layer

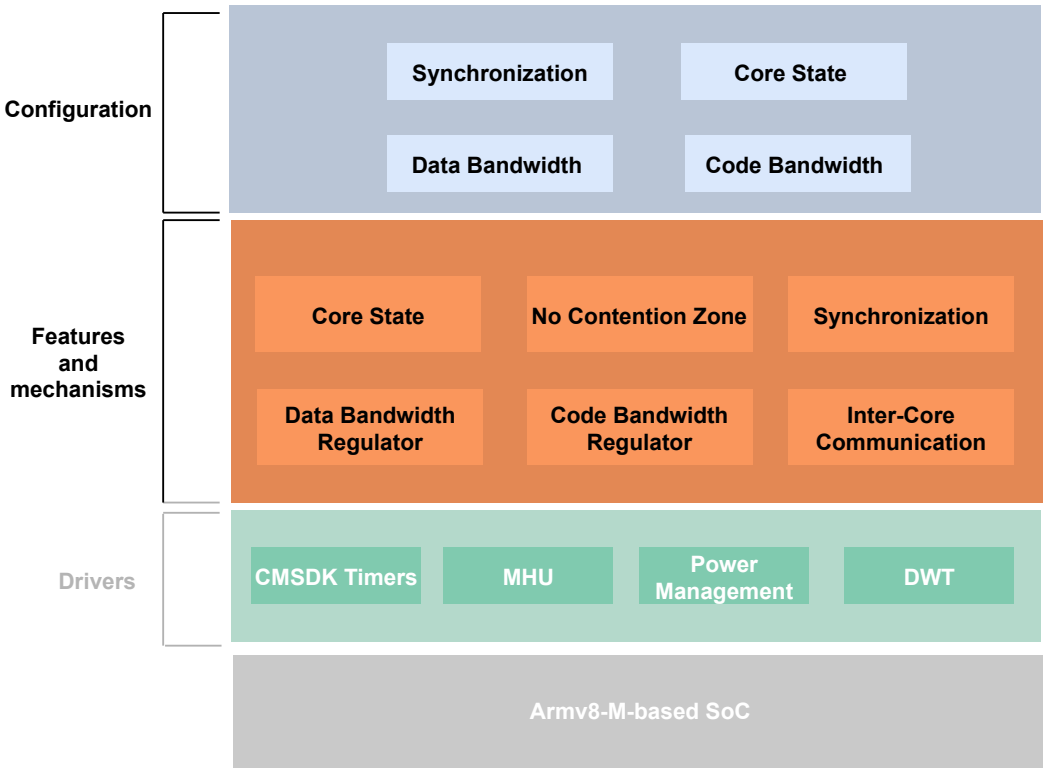


Figure 4.2: KIR Architecture: Middle and Top Layers.

This section focuses on implementing the mechanism's multiple functionalities by the use of the drivers previously described (Figure 4.2). Each module can be seen as a smaller mechanism/feature of KIR, with its independent functionality. Their implementation considers the future need for integration into uTango, so they are projected as modular, with low dependencies and, therefore, avoid extra engineering efforts.

KIR configuration follows a statically approach to configure all of its functionalities, which are defined prior to compiling time. The middle layer is composed by 6 modules and each sub-module has its own functionalities and data structure used to agglomerate the required information for the module configuration. Moreover, in each of the data structures, it is present a constant value that defines its state,i.e., enabled or disabled. The configuration layer holds a global instance named *kir_core_config* used to enabled and disable the desired functionalities.

KIR depends on synchronous exceptions to keep track of the time and delimit an execution period's size. Therefore, the KIR structures are composed of three variables: (i) *recharge_period*, that holds the user-set period size; (ii) *exception_period*, that defines the size of each tick; (iii) *exception_counter*, that is

incremented for each exception until it reaches the period size. To trigger the synchronous exception it is used the DWT Comparator 0 functionality, i.e., triggering an exception when the DWT CYCCNT matches the defined comparator 0 value. To translate seconds to clock cycles it is used the equation (4).

$$\text{Comparator 0 value } = \frac{\text{exception\_period} * \text{System Clock}}{\text{Period Time Unit}} \tag{4}$$

The *Period Time Unit* is used to determine if the exception is in seconds, milliseconds our microseconds. The formula consists of the inverse of the actual time unit. For example, one millisecond equals $10^{-}3$ seconds, so the period time unit is 1000. Therefore, the formula to determine the *Period Time Unit* is given by the equation (5):

$$\text{Period Time Unit} = \frac{1}{Time} \tag{5}$$

The exception period and time period are independently configurable. The sum of each synchronous exception determines the period execution windows. Therefore, there is a dependency between the configuration options that can lead to later problems. For example, if each exception takes 1 ms and the recharge period is 100 ms, the exception counter has to increment 100 times until the period is over. On the other hand, if each exception takes 3 ms and the recharge period is 100 ms, the exception counter has to increment 33.33 times until the period is over. The decision was made, taking into account the tradeoff between the system's resource usage and configuration flexibility.

When KIR is enabled, the Debug Monitor exception is enabled by setting the bit DEBUG_MON_EX of the DEMCR. At the entry of the Debug Monitor handler, all of the enabled DWT counters are paused and the handler can be called in two different cases: (i) the DWT comparator 0 matches the CYCCNT value; (ii) and the DWT comparator 2 triggers while watching a block of memory. Therefore, it is required to test which one of the comparators triggered the exception. If the cause is the comparator 0, the exception counter is incremented and then checked if it matches the period counter. If so, all of the counters (i.e., software and hardware) are reset and a new period begins. Otherwise, only the DWT counters are reset.

When resetting the software counters, it is required to consider if the *Core State* feature is enabled. If so, the core must enter a sleep state instead of start executing, i.e., *sleep first, execute later*. In case the exception is triggered by access to a region of memory that is being watched (i.e., comparator 2), it is executed the data bandwidth regulator code. Lastly, the DWT counters are again enabled, and the handler exited. Parts of the handler are presented in the Listing 4.1.

```
1  void DebugMon_Handler(){
2      kir_pause(&kir_core_config);
3      /* DWT Comparator 0 Exception */
4      if(DWT->FUNCTION0 & DWT_FUNC_MATCHED)
5      {
6          /* Increment the synchronous exception count */
7          kir_core_config.exception_counter++;
8          /* Core State Handler code */
9          {...}
10         /* Code Bandwidth Handler code */
11         {...}
12         /* Check if the period is over */
13         if(kir_core_config.exception_counter == kir_core_config.
       recharge_period)
14             {
15                 /* Resets all counters to start a new period (e.g.,
       exception counter, DWT counters, etcs) */
16                 kir_reset_counters(&kir_core_config);
17             }
18             /* Resets the DWT Counters only */
19             kir_code_band_reset_counters(kir_core_config.code_band);
20     }
21     /* Region of memory access match */
22     else if(DWT->FUNCTION2 & DWT_FUNC_MATCHED || DWT->FUNCTION3 &
       DWT_FUNC_MATCHED)
23         {
24             /* Code Bandwidth Handler code */
25             {...}
26         }
27         kir_go(&kir_core_config);
28 }
```

Listing 4.1: KIR Debug Monitor Handler used to track the time passed.

No Contention Zone mechanism: It depends on the inter-communication module and, therefore, the MHU handler (see Listing 4.2). Any one of the cores can ask for a *No Contention Zone*. When done so, the MHU handler is called, and the message is read from the unit. By matching the first test case, where the message is a *SleepRequest*, all of the timers and counters are paused, and the core is set to a sleep state once the interrupt exits. In the end, the core sends a *SleepConfirm*.

When the core that made the *No Contention Zone* request finishes its execution of critical code, a *WakeUpRequest* is sent to the sleeping core. When received a *WakeUpRequest*, all of the timers and counter are reset, and the mechanism enabled again.

The end of *No Contention Zone* also works as a synchronization point, if enabled, since both cores reset their counters.

```
1  void MHU0_IRQHandler(){
2      /* Reads the last received message from the MHU channel */
3      kir_sync_get_last_rec_message(kir_core_config.kir_sync);
4      if(kir_core_config.kir_sync->core_coms->received_message ==
       SleepRequest){
5          /* pause the mechanism */
6          kir_pause(&kir_core_config);
7          /* pause  synchronization */
8          kir_sync_pause(kir_core_config.kir_sync);
9          /* make the core go to sleep */
10         kir_core_state_sleep(kir_core_config.core_state);
11         /* send a sleep confirm */
12         kir_sync_sleep_confirm(kir_core_config.kir_sync);
13     }else if(kir_core_config.kir_sync->core_coms->received_message ==
       WakeUpRequest){
14             /* reset all the counters */
15             kir_reset_counters(&kir_core_config);
16             /* send a wake up confirm message */
17             kir_sync_wake_up_confirm(kir_core_config.kir_sync);
18             /* enable back synchronization */
19             kir_sync_go(kir_core_config.kir_sync);
20             /* enable the mechanism again */
21             kir_go(&kir_core_config); }
22     {...} }
```

Listing 4.2: MHU Handler of the No Contention Zone.

Core State Feature: is an individual module that consists in a data structure and a set of functions used to manipulate it. The data structure is composebe by (i) a volatile enumeration variable that changes (zero if sleep, one if active) when the state of the core changes; (ii) a constant variable that holds the sleep time defined by the user; (iii) and a counter that keeps track of how much time the core has slept. For each exception generated by the DWT comparator 0, if the core is in a sleep state (i.e., the Core State functionality is enabled), the counter is incremented until it matches the sleep time defined (Listing 4.3). If the sleep time is over, the state's core changes to active, and the core starts running, and the sleep counter is reset.

```
1 void DebugMon_Handler(){
2     {...}
3         /* Update the counter and checks if the core should switch from
      sleeping to running */
4         kir_core_state_update(kir_core_config.core_state);
5     {...}
6 }
```

Listing 4.3: Core State Debug Monitor Handler.

Code Bandwidth Regulator: is used to regulate the code usage of the core by using the DWT driver. The regulator module is composed of a data structure and a set of functions that help its configuration and manipulation. The structure holds a variable used to define the maximum instructions that the core can execute, and a counter that takes the number of instructions executed by the core at any given time.

When the mechanism inits the regulator, it sets to zero the instruction count. For each synchronous exception triggered by the DWT comparator 0, it is added the relative instruction count to the total instructions performed by the core. In case the total count is equal to or higher than the user-defined maximum number of instructions it can perform, the instruction count is set back to zero and the core set to a sleep state. Otherwise, the core continues to execute.

Since the exceptions will continue to be triggered even if the core is in a sleep state (i.e., to keep track of time passed until a new period begins), before incrementing the instruction counter, it is required to verify if the core is in a running state (see Listing 4.4).

```
1  void DebugMon_Handler(){
2      {...}
3          /* Check if the core is in running state and if the regulator is
       activated */
4          if(kir_core_config.core_state->state == active &&\
5           kir_core_config.code_band->status == on)
6          {
7              /* Add to the total instruction count the relative instruction
       count */
8              kir_code_band_set_instruction_count(kir_core_config.code_band);
9          }
10         /* Check if the instruction count has passed the threshold */
11         if(kir_core_config.code_band->instruction_count >\
12         kir_core_config.code_band->max_instruction_count)
13         {
14             /* Reset the instruction count */
15             kir_core_config.code_band->instruction_count = 0;
16             /* Change the core to a sleep state */
17             kir_core_state_sleep(kir_core_config.core_state);
18         }
19     {...}
20 }
```

Listing 4.4: Code Bandwidth Regulator Debug Monitor Handler.

Data Bandwidth Regulator: Like the Code Bandwidth, it is supported by the driver of the DWT unit. The structure created for this module holds the user-defined maximum number of memory accesses the core can perform, a counter that holds the number of accesses done to memory, the block of memory to be watched starting address, and the memory block's size. During the initialization of the regulator, it is set the memory access count to zero, the DWT comparators are configured using the DWT driver and set to trigger an exception for each write or read to the part of memory that is being watched.

When a part of the memory that is being watched has been read or written to, an exception is triggered and the data access counter incremented. In case it is reached the threshold of accesses, the core is set into a sleep state.

```
1  void DebugMon_Handler(){
2      {...}
3      /*Increments the data access counter and checks if has gone over the
       threshold*/
4      if(++kir_core_config.mem_band->mem_access_count >\
5              kir_core_config.mem_band->max_mem_access_count)
6              {
7                  /* Change the core to a sleep state */
8                  kir_core_state_sleep(kir_core_config.core_state);
9              }
10     {...}
11 }
```

Listing 4.5: Data Bandwidth Debug Monitor Handler.

Inter-Core Communication: The Inter-Core Communication module sits on top of the MHU driver. Since there are two MHUs and different mapped locations in the VTOR, one must be predefined, and, in case of change, the handler's name must also change. In this implementation, the MHU chosen was the first one (i.e., MHU0).

For the configuration of the message handling unit, it was created a structure that holds the mapped memory location of the unit, the CPU id of the core, and the selected MHU. All of them are constant from the point of definition and never change during run-time. Moreover, two volatile variables are used to know what message was sent last and the last one received. Contrary to the other resources used, the MHU only requires that each core enables its IRQ. To know the MHU channel's state, it is used a volatile variable defined in the data structure.

The objective of this module is to create an abstraction between the underlying hardware and the inter-core communication. Therefore, if another platform has a different type of mechanism to exchange messages between cores, it is only required to alter this single module.

The module defines two functions that are used to send and read messages: (i) *init* to enable the MHU interrupt; (ii) *send* function to send a message that uses the *set* function of MHU driver; (iii) and a *receive* function that uses the *status* function of the MHU driver to read the latest received message and, the *clear* function to clear the interrupt flag.

Synchronization Feature: Like in every other module, there is a structure that supports and agglomerates all of the required data for its operation. This structure contains (i) the synchronization period defined by the user; (ii) a pointer to an instance of the inter-communication structure type; (iii) and a pointer to the memory location of the selected timer. The set of messages proposed in the design (Table 3.8) are created using an enumeration, making an understandable correlation between a name and a value.

The module implements four types of functions that are required to synchronize both cores: (i) a function to init the peripherals (i.e., timer and MHU); (ii) a function to enable/disable the timer and MHU interrupt; (iii) a function pair named go/pause to only start/stop the timer but mantain the MHU interrupt enabled; (iv) and lastly, a function to send/receive messages.

The synchronization can happen at boot or during the normal execution of the application. As described in the design (Section 3.3), only one CPU0 will be in charge of doing the synchronization. Therefore, the CPU0 has to set up the timer and then enable its interrupt. Mandatorily, both cores need to enable the MHU interrupts. To determine the RELOAD value of the time, it is used the equation (6):

$$\text{Reload value} = \frac{\text{sync\_period} * \text{System Clock}}{\text{Period Time Unit}} \tag{6}$$

The MHU handler previously listed (Listing 4.2) is completed by the last two test cases introduced by the synchronization mechanism (Listing 4.6). Considering CPU0 is in charge of the synchronization and therefore will send a *SyncRequest*, CPU1 only needs to check if the received message equals it. And the same goes for CPU0, where it only needs to test if the message received is *SyncConfirm*. Therefore, to reduce the amount of test cases in the handler, it was added macros so that specific code is injected into each core executable. When CPU1 receives a *SyncRequest*, the counters are disabled and reset, a *SyncRequest* is sent, and the counters are enabled again. Furthermore, when CPU0 receives a *SyncConfim*, the mechanism proceeds to execute again, and the state of the core is updated. The update of the core state, i.e., running or sleeping, is needed to ensure that the core is set to sleep in case the KIR *Core State* feature is enabled.

```
1  void MHU0_IRQHandler(){
2      {...}
3  #if defined CORE_1
4          }else if(kir_core_config.kir_sync->core_coms->received_message ==
       SyncRequest)
5          {
6              /* Reset all of the counters and the core state */
7              kir_reset_counters(&kir_core_config);
8              /* Send a SyncConfirm */
9              kir_sync_sync_confirm(kir_core_config.kir_sync);
10             /* Launch KIR */
11             kir_go(&kir_core_config);
12         }
13 #endif
14 #if defined CORE_0
15         else if(kir_core_config.kir_sync->core_coms->received_message ==
       SyncConfirm)
16         {
17             /* Launch KIR */
18             kir_go(&kir_core_config);
19             /* Reset the sleep counter and the state of the core */
20             kir_core_state_reset_counters(kir_core_config.core_state);
21 #endif
22 }
```

Listing 4.6: Synchronization MHU Handler.

During run-time, the synchronization is maintained by creating synchronous interrupts, where it is cleared both cores counters and then enabled back again. Since there are two timers and different mapped locations in the VTOR, one must be predefined, and in case another timer is used, the handler's name must also change. In this case, it is used timer 1. To avoid pulling in an interrupt, CPU0 pauses all of the counters and then resets them. Following this, CPU0 sends a *SyncRequest* and enters a sleep state until it receives a *SyncConfirm*. When received a *SyncConfirm*, the core restores its reset state, i.e., sleeping or running. The *SyncConfirm* is dealt with in the MHU interrupt (see Listing 4.6). Before exiting the interrupt, the timer's flag needs to be cleared, or the core will enter an infinite loop interrupts.

```
1  void TIMER1_IRQHandler(void)
2  {
3      /* clear the Timer flag */
4      cmsdk_timer_clear_irq_flag(kir_core_config.kir_sync->Timer);
5      /* Pause KIR */
6      kir_pause(&kir_core_config);
7      /* Reset all of the counters and the core state */
8      kir_reset_counters(&kir_core_config);
9      /*Send a SyncRequest */
10     kir_sync_sync_request(kir_core_config.kir_sync);
11     /*Enter a sleep State*/
12     kir_core_state_sleep(kir_core_config.core_state);
13 }
```

Listing 4.7: Syncrhonization CSMDK Timer Handler.

Configuration: The configuration layer creates the abstraction between the logic of KIR and the way it is configured. Therefore, for KIR to work correctly, it is required to set up the: (i) Period Time unit; (ii) and configuration of KIR global instance named kir_core_config. In this scenario, there are three possible values for the time unit: (i) Seconds; (ii) Milliseconds; (iii) Microseconds, and they are set by changing the PERIOD_TIME_UNIT macro (see Listing 4.8).

```
1 #define S  1
2 #define MS 1000
3 #define US 1000000
4
5 #define PERIOD_TIME_UNIT MS
```

Listing 4.8: Time Unit definition.

The Listing 1 in the Appendix 6.2 is an example of the configuration for the KIR mechanism running in CPU0, with a time period of 100 ms and with each synchronous exception taking place every 1 ms. The synchronization is done every 1s by the use of timer 1 and the MHU0. The *Core State* feature and the data bandwidth, in this case, are turned off while the code regulator is enabled with a bandwidth of 2500000 instructions per period (100ms).

4.2.1 Integrating KIR in uTango

uTango, at the time of writing, supports two boards, namely; (i) Arm Musca A1; (ii) and Arm Musca B1. uTango runs in the secure world while all of the applications run in the non-secure world, i.e., NSVW. Each of the applications is built separately and later linked with the executable of the secure kernel. Following this, each of the applications also need to be configured (Listing 4.9) in a uTango flat file, where it is defined the vector table memory location, stack pointer starting location, entry point of the application (i.e., reset handler), and the number of SAU regions to be configured. Each region needs to specify the start address and the size of the memory. It is also declared the interrupts that are going to target the non-secure world by using its IRQ number.

```
1    {
2        .id = 0,
3        .vect_base_addr = 0x00208000,
4        .init_sp = 0x20010000,
5        .entry_point = 0x00208145,
6        .n_regions = 4,
7        .w_regions = (world_regions_t[])
8        {
9            {
10               .rgn_base_addr = 0x00080000,
11               .rgn_size = 0x0000A000
12           },{
13               .rgn_base_addr = 0x20008000,
14               .rgn_size = 0x8000
15           },{
16               .rgn_base_addr = 0x40102000,
17               .rgn_size = 0x1000
18           },{
19               .rgn_base_addr = 0x00208000,
20               .rgn_size = 0x0000A000 }
21       },
22       .n_irqs = 1,
23       .enabled_irqs = (irqs_list_t[])
24       {CMSDK_TIMER0_IRQn}, }
```

Listing 4.9: Configuration example of a NSVW.

uTango has a low memory footprint and aims to contention-free scenario; thereby during boot, all of its data and code can be copied from the QSPI Flash memory to its core TCM (e.g., iSRAM0 for CPU0 and iSRAM3 for CPU1). This implementation detail ensures that almost all uTango kernel instructions and memory accesses takes 1-2 clock cycles and that there is no additional burden due to wait states and bus/memory stalls (performance) and that code and data are never cached (security).

To finish the system's booting process, all the fault handlers are enabled, the interrupts configured to target Secure or Non-Secure depending on the resources needed by the non-secure applications, and the *sleep mode* of the CPU is set. Also, the TCB is initialized for each application, so their execution can be restored after a context-switch.

After the booting process, the configuration of uTango (e.g., SysTick) and the TrustZone mechanisms (e.g., SAU, MPC, and PPC) of the non-secure worlds is set. The configuration phase considers each of the application configurations saved in a constant array structure named *world_cfg_t*. The size of the array tells the number of existing applications, and it is defined by a macro named UTANGO_WORLDS existent in the configuration module (Listing 4.10). Lastly, the number of interrupts in each non-secure world is defined by the UTANGO_MAX_IRQs.

To track the time passed and preempt the running non-secure worlds, the system timer is used, i.e., the SysTick timer. A macro named UTANGO_TICK defines the time quantum (i.e., the tick) of the system; thereby, the SysTick reload time is calculated by the equation (7):

$$\text{SysTick load value} = \frac{\text{UTANGO\_TICK} * \text{System Clock}}{\text{Period Time Unit}} \tag{7}$$

The SysTick is initialized during the configuration, and later, before jumping to the non-secure world, is enabled. The starting NSVW is the first one of the *world_cfg_t* array. uTango uses a pointer named *rworld_ctx_ptr* to know which one of the worlds is executing at any given time.

In the SysTick handler, all of the interrupts are disabled and, the CPU register and NVIC configuration of the running application are saved. Next, the SAU regions are reconfigured and, the NVIC and the CPU registers set accordingly to NSVW last state. In the end, the interrupts are enabled and the newly scheduled NSVW starts executing. In case there is only one NSVW running, it proceeds its execution after exiting the SysTick handler.

```
1 #define  UTANGO_TICK          10 /* ms */
2 #define  UTANGO_WORLDS        1
3 #define  UTANGO_MAX_IRQS      4
```

Listing 4.10: Configuration example of uTango.

Single-core to Multi-core: To follow the design proposed in Section 3.3.1, where uTango will run in a multi-core AMP configuration, it was required to port it to a multi-core scenario. Therefore, tweaks were done to uTango in order to have the system boot in both cores. Firstly it was required to define the workload distribution between the different cores. In this case, the attribution of each non-secure application to each core was done incrementally; thereby, the first application is run in the CPU0, the second in the CPU1, the third in CPU0, and so on.

During the boot process, both cores initialize all of the non-secure applications TCBs, memories, interrupts, etc. Additionally, CPU0 is required to set CPU1 VTOR and then enable it. Moreover, another macro named MULTICORE was added to the configuration source file of uTango and later defined in the makefile; thereby, uTango can be compiled and executed in a single-core scenario without being required to always wake up the second core.

Modifications to the uTango linkerfile and build system were also required in order to produce two executables. Since there is no type of memory virtualization in the system, it is impossible to have a memory overlap between different executable' data and code locations. Therefore, each cores' executable (i.e., uTango) has its own linkerfile (e.g., core_0.ld and core_1.ld).

Considering KIR was implemented in a modular way, the integration requires minor changes to the code base of KIR. Hereafter, KIR stops from being a solo bare-metal mechanism and becomes part of the uTango kernel and, therefore, it interacts with it.

In the design (Section 3.3.1), it was proposed that code and data bandwidth target each non-secure application and, the rest of the features, such as *Synchronization* and *Core State*, are transversal to all worlds, i.e., target the core and not the worlds. Thus, it is necessary to alter some uTango (e.g., *world_cfg_t*) and KIR structures.

KIR Regulators: Since each application has its boundaries, the uTango world_cfg_t structure was modified to incorporate pointers to instances of KIR code and data bandwidth. Moreover, restructuring KIR results is done by altering its data structure; instead of having the code and data bandwidth pointers, it embeds a pointer to a structure of type world_cfg_t (Listing 4.11).

```
typedef struct world_cfg {
    {...}
    kir_mem_band *mem_band;
    kir_code_band *code_band;
} world_cfg_t; /* uTango structure */

typedef struct {
    {...}
    const world_cfg_t *worlds_cfg;
}kir_core; /* KIR structure */
```

Listing 4.11: uTango and KIR data structure changes

The change to the KIR structure forces a shift in the KIR global module functions and the handlers but does not require changing its underlying, such as the code and data bandwidth regulator modules. Instead of directly accessing the counters, the world running must be specified by passing its *id* as a parameter.

For each of the uTango worlds to be targeted instead of the entire core, it is also needed to change the way it is accessed the altered KIR data structure. In other words, the KIR data structure no longer embeds pointers to the regulators but instead to the existing worlds. Therefore, it is used the *world_cfg_t* and the id of the running world given by the *rworld_ctx_ptr* to update the world's regulators. Listing 4.12 shows an example of the world's counter being updated.

```
void DebugMon_Handler(void) {
    volatile uint32_t world_id  = rworld_ctx_ptr->world_id;
    {...}
    kir_code_band_reset_counters(kir_core_config.worlds_cfg[world_id].
    code_band);
    {...}
}
```

Listing 4.12: KIR handler example after integration in uTango

Synchronization feature: For uTango integration, the synchronization no longer uses a timer but instead uses the already existent scheduling points of each world. Therefore, changes to the handler and the KIR synchronization module were required. For more flexibility and, therefore, have a different timing between synchronization and scheduling points, the *sync_period* variable present in the KIR synchronization data structure is now used as a counter. With this change it is possible to have multiple worlds scheduling

before a synchronization occurs. For example, if the UTANGO_TICK is 10 ms and the synchronization period is 500 ms, it is required 50 scheduling points until a synchronization is triggered. The *sync_period* is calculated by using the equation (8):

$$\text{Sync Period} = \frac{\text{Sync Period}}{\text{UTANGO TICK}} \tag{8}$$

To integrate the KIR synchronization feature, it is required to alter parts of the scheduler. In uTango, the handler is all written in Arm assembly Thumb code. Therefore, to call the KIR functions to check if it is required a synchronization, according to the ABI [92], it is necessary to save all the registers from r4 to r11 and the link register. At the beginning of the handler, all the interrupts are disabled and then called the context switch pause function. At the end of the SysTick handler, before enabling back the interrupts, the context switch go function is called.

No Contention Zone mechanism: Lastly, the *No Contention Zone* continues to target a specific part of the running non-secure application. So this means this section of code the non-secure side must access the existent functions in uTango by making calls to the secure state. Specific attributes are added to the function so that they are considered Non-Secure Callable and placed in a region of memory the non-secure side can access, as explained in the State of the Art (Section 2.1.1.6).

During the *No Contention Zone* both core's disable their scheduler until critical code finishes its execution. Therefore, the no contention zone start and end functions required changes to stop and start the SysTick, respectively, so that the core can run without any interruption. Furtheremore, the MHU Handler also needed to be altered to stop and start the SysTick when receiving a *SleepRequest* and *WakeUpRequest*, respectively.

Chapter 5

Experimental Results

The previous chapter described the implementation of KIR and its porting to the uTango TEE. Consequently, this chapter aims to test the developed mechanism under the same conditions as the ones used in the preliminary results (Section 3.2).

The chapter is partitioned into two sections where the first one (i.e., Section 5.1) is presented the benchmark results of KIR in a bare-metal scenario, while the Section 5.2 is described the KIR test results after its integration in uTango. Regarding each one of the sections, it is required to repeat the baseline results previously done in the preliminary results due to the presence of KIR, i.e., KIR trades performance for predictability; therefore, the benchmark baseline results are expected to be different than the previous ones. Thus, it was conducted experiments for multiple KIR configurations, and then it was collected the contention results and to be compared with the baseline ones.

5.1 KIR Benchmarking

This subsection presents a detailed analysis of the KIR in a bare-metal scenario. The goal is to provide a set of empirical observations of KIR mechanism and features, and prove that it can help improve the determinism of the system. This section has five subsections: (i) first and second are used to describe the code bandwidth regulators tests; (ii) third and four show the data bandwidth regulator tests; (iii) and the fifth is used to present the *No Contention Zone* mechanism test. All experiments were conducted using the same compiling optimizations and memory layout previously used in the preliminary results: (i) when it is used the eSRAMs for code, the optimization flag is -O3; (ii) and, if it is used the iSRAM for code placement, the optimization flag is -Os, so that the performance degradation (P.D.) can be comparable. The cache is disabled in all of the benchmarks and each core runs at a frequency of 50Mhz.

In the subsection 5.1.1 is described all the steps folowed to find out the best possible exception period for KIR, so that the DWT counters values do not become too much imprecise and, and the same time, do not introduce jitter into the system caused by the exception overhead. Moreover, it is tested how many instructions the core can perform while running solo in a specific execution window (e.g., 100ms). After collecting the maximum bandwidth (x), two lower values, i.e., y and z, are selected, and three benchmarks are run to collect the baseline results under the effects of regulation. Moreover, the subsection 5.1.2 is composed of multiple Coremark benchmarks with both cores running KIR. For each experiment, it is enabled different features of KIR until the best possible results are achieved.

The subsection 5.1.3 presents the collected baseline results of each one of the CPUs when running solo with the data bandwidth regulator. For each run, it is used a different data bandwidth level. The bandwidth levels are selected based on a benchmark that collects the maximum bandwidth available in a particular execution window (e.g., 100ms). Moreover, the subsection 5.1.4 is rerun the Coremark when both cores have KIR data bandwidth regulators enabled. The results help determine if this regulator produces any improvements in the system predictability.

Lastly, in subsection 5.1.5, it is tested the *No Contention Zone* mechanism. One of the cores was selected as the test subject while the other runs normally. In the end, it is outputted and analyzed the results produced by both cores.

5.1.1 Code Bandwidth Regulator Baseline Results

The benchmarking process starts by finding the best possible period for the synchronous exception. The period chosen had to be the one that introduced none to almost none jitter into the system and, at the same time the must have the lowest possible value since all DWT counters, except the CYCCNT, are 8-bit precise; thereby, after overflowing, they are automatically reset and continue to count. Therefore, the higher the exception period, the less the precision when measuring the core's relative instructions count.

Table 5.1 depicts the Coremark results for multiple exception periods when the CPU0 runs from the eSRAM0. The data segment was placed in its TCM bank, which for CPU0 is the iSRAM1.

Iterations	Time	Exception Period
1500	+60s	0.000001s
1500	18.59s	0.00001s
1500	11.43s	0.0001s
1500	10.75s	0.001s
1500	10.68s	0.01s

Table 5.1: Coremark results for different KIR exception periods.

Table 5.1 shows that the lower the exception period, the higher the time required to complete 1500 iterations of Coremark. This is a result of the overhead introduced by the Debug Monitor exception. The last results, i.e., 1ms and 10ms, proved to be the most effective and efficient timings for the synchronous exception. Additional experiments were conducted to decide if the exception period should be 1ms or 10ms. The results showed that the 1ms period, as expected, was the most precise and therefore the chosen one, although introducing an acceptable overhead.

Table 5.2 depicts the Coremark baseline results for each one the cores when running from the eSRAMs with KIR code bandwidth regulator enabled. The code segment for CPU0 is placed in the eSRAM0 and, the code segment for CPU1 is placed in the eSRAM3. The data segment is placed in each core TCM bank, which for CPU0 is the iSRAM bank one (i.e., iSRAM1), and for CPU1 is bank three (i.e., iSRAM3). The three selected bandwidth values were based on the maximum number of instructions a core could perform in 100ms (i.e., size of the execution window). The period size chosen required to be higher than the exception period, i.e., synchronous exception, but not higher to the point of overflowing the software instruction counters.

					CPU0		CPU1	
CPU0		CPU1			Cache Disabled			
code	data	code	data	Iterations	Time(s)	Code Bandwidth	Time(s)	Code Bandwidth
eSRAM0	iSRAM1	eSRAM3	iSRAM3	1500	10.75	5000000	40.10	5000000
					14.13	3750000	52.86	3750000
					21.04	2500000	79.14	2500000

Table 5.2: CPU0 and CPU1 KIR Code Bandwidth Coremark baseline results.

Table 5.2 shows that, for a period size of 100ms and an exception period of 1ms, the higher the bandwidth, the lower the time it will take the core to complete 1500 iterations of Coremark. Moreover, the results show that the maximum code bandwidth available in 100 ms is close to 5000000 instructions. Therefore, the time results are almost equal to when the core executes without any regulation.

When using the KIR mechanism, it is expected for the CPU to have an higher P.D. since the CPU is limited when executing. For example, CPU0 running KIR solo and limited to 2500000 instructions has a base P.D. of -49% (21.04s). This outcome has significancy in terms of performance, but no evidence with regard to predictability. Therefore, to analyze the core contention results when using KIR regulators, it it should be taken into account the base performance degradation, i.e., relative performance degradation (R.P.D.) to the baseline timings. The R.P.D. shows how much jitter was introduced into the system due to the contention derived from the shared bus to the external memories infrastructure. Table 5.3 depicts each core's performance degredation for the three code bandwidth limits when running solo.

CPU0			CPU1		
Time(s)	Code Bandwidth	Performance Degradation (%)	Time(s)	Code Bandwidth	Performance Degradation (%)
10.75	5000000	- 1	40.10	5000000	- 2
14.13	3750000	- 24	52.86	3750000	- 26
21.04	2500000	- 49	79.14	2500000	- 51

Table 5.3: Coremark KIR CPU0 and CPU1 base performance degredation with KIR Code Bandwidth Regulator.

The P.D. of each CPU, is calculated in comparison with the core baseline result when each CPU is running alone running without the interference of KIR, i.e., 10.67s for CPU0 and 38.95s for CPU1. When running without KIR, the CPU0 and CPU1 have a P.D. of -43% and -14%, respectively. Therefore, the predictability of the system increases if the R.P.D. of each core, when running concurrently and without the regulators, is less than the performance degradation of the core, i.e., -43% (CPU0) and -14% (CPU1). The R.P.D. is calculated by using the baseline results of each core when running solo with the KIR code bandwidth regulator (e.g., 21.04s for CPU0 and 79.14s for CPU1 when limited to 2500000 instructions).

5.1.2 Code Bandwidth Regulator Contention Results

This subsection explains the results achieved after re-running the previous experiment but with the crucial difference of both cores now running concurrently. The aim is to stress the memory infrastructure and verify if the degradation of each cores diminishes. Therefore, if the results show a lower variability in relation to the baseline results, when running regulated, it can be concluded that KIR has improved the system's predictability.

Following this memory infrastructure, three main experiments were drawn and performed per KIR mechanism or feature state (i.e., enabled or disabled). These configuration layouts are the following:

1) In the first experiment, CPU0 runs freely while CPU1 has the code bandwidth regulator enabled. The Coremark benchmark runs three times with the three different bandwidth limits (i.e., 5000000, 3750000, and 2500000 instructions per period).

2) In the second experiment, the code bandwidth regulator is enabled in both cores. This experiment is split into two different experiments according to the KIR *Core State* feature (i.e., enabled or disabled).

3) In the third experiment, both cores run with the code bandwidth regulators enabled and with the KIR synchronization feature enabled. This experiment is tear down into two smaller experiments according to the KIR *Core State* feature (i.e., enabled or disabled).

Throughout the multiple iteration of the KIR configurations, both cores are running concurrently and fetching code from the eSRAMs. The memory layout is the same as the one used in the baseline experiment, i.e., the code segment for CPU0 is placed in the eSRAM0 and the code segment of CPU1 is placed in the eSRAM3. The data segment is placed in each core TCM bank, which for CPU0 is the iSRAM bank one (i.e., iSRAM1), and for CPU1 is the iSRAM bank three (i.e., iSRAM3).

Experiment #1 (Code Bandwdith Regulator in CPU1): Table 5.4 results show that the CPU0 benchmark timing results decrease substantially when the CPU1 has less code bandwidth, at the cost of CPU1 performance and predictability. For CPU0 it is expected that the R.P.D. to be closer to zero; while for CPU1, it is expected for the R.P.D. to be the closest possible to its expected P.D. presented in preliminary results, i.e., -14%. The reason is that CPU0 is running without any regulation while CPU1 is limited; therefore although CPU1 takes more time to complete the 1500 iterations of Coremark, it still maintains the same R.P.D. of around - 14%; thereby the CPU1 shows no improvements in this type of scenario.

CPU0				CPU1				
Iterations	Time(s)	P.D.(%)	R.P.D.(%)	Iterations	Time(s)	Code Bandwidth	P.D.(%)	R.P.D.(%)
1500	18.03	- 41	- 41	1500	46.84	5000000	- 17	- 14
	15.55	- 31	- 31		61.78	3750000	- 37	- 14
	13.57	- 21	- 21		92.25	2500000	- 58	- 14

Table 5.4: Coremark CPU0 results while CPU1 is limited in Code Bandwidth. P.D. refers to performance degradation. R.P.D refers to relative performance degradation.

Experiment #2 (Synchronization Disabled): Table 5.5 depicts the Coremark results obtained for each CPU is running concurrently and fetching code from the eSRAMs. In this benchmark, both cores have the KIR code bandwidth regulators enabled and non of the other feature enabled, i.e., *Core State* and *Synchronization*.

	CPU0				CPU1			
Iterations	Time(s)	Code Bandwidth	P.D. (%)	R.P.D. (%)	Time(s)	Code Bandwidth	P.D. (%)	R.P.D. (%)
1500	18.21	5000000	- 41	- 41	46.90	5000000	- 17	- 14
	16.22		- 34	- 34	61.67	3750000	- 37	- 14
	13.60		- 21	- 21	92.33	2500000	- 58	- 14
	25.33	3750000	- 58	- 44	44.92	5000000	- 13	- 11
	21.27		- 50	- 34	59.34	3750000	- 34	- 11
	18.51		- 42	- 24	88.82	2500000	- 56	- 11
	36.15	2500000	- 70	- 42	43.09	5000000	- 10	- 7
	30.39		- 65	- 31	56.88	3750000	- 31	- 7
	26.85		- 60	- 21	85.12	2500000	- 54	- 7

Table 5.5: Coremark contention results with KIR Code Bandwidth regulators only.

To undestand if the system's predictability has improved, it was decided to compare the Coremark results with each core's baseline results when running with the code bandwidth regulator. By doing that, it is taken into consideration the CPU base degradation introduced by KIR. For example, CPU1 R.P.D. in the top three contention results (i.e., 46.9s, 61.67s, and 92.33) is around -14%. This result, as expected,

matches the CPU1 degradation of the core when running concurrently with CPU0 without KIR (i.e., -14%). Furthermore, when CPU0 starts to be limited to 3750000 instructions, the results show that the R.P.D. of CPU1 decreases to -11% and, later too -7% when CPU0 is limited to 2500000. For CPU0, most of the tests' R.P.D. is -43% when CPU1 is limited to 5000000 instructions, then drops to 33% when CPU1 is limited to 3750000 instructions and achieves the best result of -21% when the CPU1 is limited to 2500000 instructions. Therefore, the results show that the system predictability has significantly improved but there is still jitter in the system, i.e., -21% for CPU0 and -7% for CPU1, when both cores are limited to 2500000 instructions.

Table 5.6 depicts the benchmark results when both cores have the KIR code bandwidth regulators and *Core State* feature enabled. When CPU1 is limited to 3750000 instructions per period, the core sleeps for 25ms before starting its execution and, when limited to 2500000 instructions, the core sleeps for 50ms until it starts executing.

	CPU0				CPU1			
Iterations	Time(s)	Code Bandwidth	P.D. (%)	R.P.D. (%)	Time(s)	Code Bandwidth	P.D. (%)	R.P.D. (%)
1500	18.55	5000000	- 42	- 42	47.81	5000000	- 19	- 16
	16.22		- 34	- 34	63.11	3750000	- 38	- 16
	13.78		- 23	- 22	94.77	2500000	- 59	- 16
	24.38	3750000	- 58	- 42	45.71	5000000	- 15	- 12
	21.34		- 50	- 34	60.28	3750000	- 35	- 12
	18.31		- 42	- 23	90.67	2500000	- 57	- 12
	37.85	2500000	- 70	- 44	43.37	5000000	- 10	- 8
	30.89		- 65	- 32	57.52	3750000	- 32	- 8
	26.85		- 60	- 22	86.45	2500000	- 55	- 8

Table 5.6: Coremark contention results with KIR Code Bandwidth regulators and Core State active on CPU1.

The same approach to analyze the results is as previously, i.e., compare the results of each CPU with the baseline results when running with the code bandwidth regulator. For example, CPU1 R.P.D. in the top three contention results (i.e., 47.81s, 63.11s, and 94.77s) is around -16%. Furthermore, when CPU0 starts to be limited to 3750000 instructions, the results show that the R.P.D. of CPU1 decreases to -12% and, later to -8% when CPU0 is limited to 2500000. For CPU0, the R.P.D. in most of the tests is -43% when CPU1 is limited to 5000000 instructions, then drops to -33% when CPU1 is limited to 3750000 instruction and reaches the best result of -21% when the CPU1 is limited to 2500000 instructions.

The results with the *Core State* feature show a slight increase in most of the R.P.D of both CPUs due to the non existence of synchronization. Therefore, the strategy *sleep first, execute later* can cause more overlap of the execution windows of both CPUs, i.e., a worst overall system performance and predictability. Overall, although with increased jitter in the system, both CPUs still show increased predictability since the R.P.D. is lower than when the cores are running without regulation.

Experiment #3 (Synchronization Enabled): Table 5.7 depicts the Coremark results obtained for each core with the code regulators and *synchronization* feature enabled. The synchronization points were set to be at every 1s. The size between synchronizations were chosen to be a multiple of the synchronous exception (1ms) and higher than the size of the execution window (100ms). Moreover, tests showed that after around 10 periods, a synchronization is required, and therefore the chosen 1s.

The *Synchronization* feature forces that each core to execute or sleep at the same time. Therefore, it is expected that the cores degradation equals the contention results, i.e, around -43% for CPU0 and -14% for CPU1, when the limited core has less or equal bandwidth than the other core and improves when it has more bandwidth. For example, CPU0 in the fourth and fifth benchmark (i.e., 23.7s and 23.91s) has a R.P.D. of -41% when CPU1 has more, or the same code bandwidth (i.e., 5000000 and 3750000), but has a decrease in its degradation when CPU1 is limited to 2500000 instructions, i.e., -28%. In the last three benchmarks, CPU0 constantly has a R.P.D. of around -40% since it is always executing when the CPU1 is also executing. The same logic applies to CPU1; when it has less or equal code bandwidth than CPU0, it has a degradation of around -14% and only decreases to -7% when CPU0 has less code bandwidth, i.e., last two benchmarks (43.03s and 57.26s).

The overall results show that just using the synchronization feature harms the system performance and predictability since both CPUs are regulated and running in overlapped periods.

	CPU0				CPU1			
Iterations	Time(s)	Code Bandwidth	P.D. (%)	R.P.D. (%)	Time(s)	Code Bandwidth	P.D. (%)	R.P.D. (%)
1500	18.23	5000000	- 41	- 41	47.27	5000000	- 18	- 15
	16.09		- 34	- 33	61.81	3750000	- 37	- 14
	13.72		- 22	- 22	91.80	2500000	- 58	- 14
	23.70	3750000	- 55	- 40	45.22	5000000	- 14	- 11
	23.91		- 55	- 41	61.11	3750000	- 36	- 14
	19.50		- 45	- 28	92.20	2500000	- 58	- 14
	35.59	2500000	- 70	- 41	43.03	5000000	- 9	- 7
	35.19		- 70	- 40	57.26	3750000	- 32	- 8
	34.46		- 69	- 39	90.61	2500000	- 57	- 13

Table 5.7: Coremark contention results with KIR Code Bandwidth regulators and Synchronization enabled.

Lastly, Table 5.8 depicts the Coremark results obtained for each core running concurrently and with the combination of all KIR features, i.e., the code bandwidth regulators, *Synchronization* and *Core State* in CPU1. When CPU1 is limited to 3750000 instructions per period, the core sleeps for 25ms before starting its execution and, when limited to 2500000 instructions, it sleeps for 50ms until it starts executing.

Combining the *Core State* and *Synchronization* feature, each CPU has zones of no contention when the code bandwidth differs from 5000000. Therefore it is expected that each cores predictability increase when limited to 3750000 or 2500000 instructions. The best possible result is expected to be when both cores are limited to 2500000 instructions since both cores will run when the other core is sleeping.

For example, when CPU0 and CPU1 are both limited to 3750000 instructions, when compared to base performance degradation, i.e., -41% for CPU0 and -14% for CPU1, show improvements (i.e., -29% for CPU0 and -8% for CPU1). Although with the same code bandwidth, since they are synchronized and in opposite phases of execution, they only share resources for a very short ammount of time of the KIR-imposed execution window. Moreover, the best outcome is presented in the last benchmark (i.e., 21.84 for CPU0 and 80.94s for CPU1) when both cores are limited to 2500000 instructions, i.e., CPU0 shows a R.P.D. of around -4% and CPU1 of around -2%. Such low-performance is achieved because both CPUs execute while the other is sleeping.

	CPU0				CPU1			
Iterations	Time(s)	Code Bandwidth	P.D. (%)	R.P.D. (%)	Time(s)	Code Bandwidth	P.D. (%)	R.P.D. (%)
1500	18.69	5000000	- 43	- 43	47.76	5000000	- 18	- 16
	15.91		- 33	- 32	63.31	3750000	- 38	- 17
	13.69		- 22	- 21	94.75	2500000	- 59	- 16
	24.32	3750000	- 56	- 42	45.62	5000000	- 15	- 12
	19.80		- 46	- 29	59.65	3750000	- 35	- 11
	16.63		- 36	- 15	86.35	2500000	- 55	- 8
	36.20	2500000	- 71	- 42	43.44	5000000	- 10	- 8
	27.01		- 61	- 43	56.21	3750000	- 31	- 6
	21.84		- 51	- 4	80.94	2500000	- 52	- 2

Table 5.8: Coremark contention results with KIR Code Bandwidth regulators, Synchronization and Core State enabled.

5.1.3 Data Bandwidth Regulator Baseline Results

The benchmarking of the data memories start by gathering the baseline results of the core when the data bandiwdth regulators are enabled. As in the code bandwidth, KIR decreases the performance of the system but with the expectance of a higher predictability. Moreover, due to the invasive nature of watching memory, a higher performance degradation for each of the cores is expected when regulated. Following the same memory layout as the one used in the preliminary benchmarks, each application has its data placed in a shared iSRAM bank, i.e., iSRAM2, and its code in their own TCM. Therefore, CPU0 uses iSRAM0 for its code, and CPU1 uses iSRAM3. Throughout the iSRAM memory infrastructure benchmarking, it is used an compiler optimization of -OS, so that the code can fit into one iSRAM bank.

In this benchmark, the memory region to be watched is an array used by Coremark to store and read the results throughout each iteration. Therefore it is guaranteed that it is read and written to that zone of memory while being watched. An external benchmark result demonstrated that by giving around 10000 data accesses to the CPU, it would complete the Coremark 1500 iterations without any performance degradation. Therefore, it was selected three inferior values as data bandwidth limits, i.e., 1000, 2000, and 3000 accesses.

					CPU0		CPU1	
CPU0		CPU1			Cache Disabled			
code	data	code	data	Iterations	Time(s)	Data Bandwidth	Time(s)	Data Bandwidth
iSRAM0	iSRAM2	iSRAM3	iSRAM2	1500	94.27	1000	94.90	1000
					47.76	2000	48.03	2000
					32.26	3000	32.43	3000

Table 5.9: CPU0 and CPU1 KIR Data Bandwidth baseline results.

Table 5.9 shows that, for a period of 100ms, the higher the data bandwidth, the lower the time it will take the core to complete 1500 iterations of Coremark; therefore the lower the P.D. of each core.

Moreover, just as in the code bandwidth regulators, it is expected that the CPUs show an higher performance degradation since it is being limited in terms of accesses to memory. For example, CPU0 running KIR solo and limited to 1000 data accesses has a base performance degradation of -79.65% (94.27s). Again, it is vital to consider that the core has a decrease of performance, but not predictability. Therefore, the contention results need to be again compared to the baseline results when the core is under regulation, i.e., relative performance degradation. The relative performance degradation highlights how much jitter was introduced into the system due to the contention derived from the sharing the same iSRAM bank for data.

Table 5.10 depicts each core's performance degredation for the three code bandwidth limits when running solo.

CPU0			CPU1		
Time(s)	Data Bandwidth	Performance Degradation (%)	Time(s)	Data Bandwidth	Performance Degradation (%)
94.27	1000	- 79.65	94.90	1000	- 73.14
47.76	2000	- 59.84	48.03	2000	- 46.93
32.26	3000	- 40.55	32.43	3000	- 20.99

Table 5.10: KIR CPU0 and CPU1 base performance degredation with KIR Data Bandwidth Regulator.

5.1.4 Data Bandwidth Regulator Contention Results

Table 5.11 depicts the Coremark results obtained for each core running concurrently and fetching code from the iSRAMs. In this benchmark, both cores have the KIR data bandwidth regulators enabled. The memory layout is the same as in the baseline experiment, i.e., the code segment for CPU0 is placed in the iSRAM0, and the code segment of CPU1 is placed in the iSRAM3. The compiler is configured with optimization flag -OS, and the platform is set up with both CPUs running at 50 Mhz frequency.

The overall performance degradation is calculated based on the core baseline result when running solo without KIR, i.e., for CPU0 is 19.18s and for CPU1 is 25.49; thereby CPU0 had a base degradation of -3.85% and CPU1 a base degradation of -6.25%. The predictability of the system increases if the relative performance degradation of each core is less than the performance degradation, i.e., -6.25% (CPU0) and -3.85%(CPU1).

	CPU0				CPU1			
Iterations	Time(s)	Data Bandwidth	P.D. (%)	R.P.D. (%)	Time(s)	Data Bandwidth	P.D. (%)	R.P.D. (%)
1500	94.36	1000	- 79.67	- 0.09	94.92	1000	- 73.14	- 0.02
	94.36		- 79.67	- 0.09	48.05	2000	- 46.95	- 0.04
	94.36		- 79.67	- 0.09	32.44	3000	- 21.42	- 0.03
	47.79	2000	- 59.87	- 0.06	94.92	1000	- 73.14	- 0.02
	47.86		- 59.92	- 0.21	48.06	2000	- 46.96	- 0.06
	47.78		- 59.86	- 0.04	32.46	3000	- 21.47	- 0.09
	32.29	3000	- 40.60	- 0.09	94.92	1000	- 73.14	- 0.02
	32.31		- 40.64	- 0.15	48.06	2000	- 46.96	- 0.06
	32.29		- 40.60	- 0.09	32.44	3000	- 21.42	- 0.03

Table 5.11: Coremark contention results with KIR Data Bandwidth results with Core State enabled but Synchronization disabled.

Table 5.11 demonstrates that by limiting each of the core data accesses, the R.P.D. of each core is significantly reduced in all of the cases (e.g., CPU1 base performance degradation is -6.25% without KIR but has a relative performance degradation of -0.02% when limited to 1000 data accesses). Although the results show that the R.P.D. has decreased, i.e., lower than the performance degredation when the cores

run without regulation, it is not enough to guarantee that the system is contention-free when sharing the same iSRAM due to the nature the memories used, i.e., TCMs.

5.1.5 No Contention Zone Results

Table 5.12 depicts the Coremark results obtained for each core running concurrently and fetching code from the eSRAMs. In this benchmark, only CPU1 has the code bandwidth regulator enabled, and CPU0 uses the KIR *No Contention Zone* mechanism. The code segment for CPU0 is placed in the eSRAM0 and, the code segment for CPU1 is placed in the eSRAM3. The data segment is placed in each core TCM bank, which for CPU0 is the iSRAM bank one (i.e., iSRAM1), and for CPU1 is bank three (i.e., iSRAM3). The compiler is configured with optimization flag -O3 and the platform is set up with both CPUs running at 50 Mhz frequency.

	CPU0				CPU1			
Iterations	Time(s)	Code Bandwidth	P.D.(%)	R.P.D.(%)	Time(s)	Code Bandwidth	P.D.(%)	R.P.D.(%)
1500	10.67	Disabled	0	0	46.10	5000000	- 16	- 13

Table 5.12: KIR No Contention Zone Results.

Table 5.12 shows, as expected, a performance degradation of 0% for CPU0 since it is running freely and without CPU1 using any of the resources. CPU1 has around the same relative performance degradation when running in a multi-core scenario with around the maximum available code bandwidth (i.e., -13%). The timing result is not higher since Coremark does not start before the CPU0 asks for a *No Contention Zone*. Therefore, the expected result of CPU1 should be around 57.51s, where 10.67s would derive from the sleep time while CPU0 is executing its 1500 iterations of coremark.

5.1.6 Discussion

In the past sections, it was presented the results of the mechanism under different scenarios where it is first analyzed the impact of the code and data bandwidth regulators and the *No Contention Zone* mechanism. Throughout the benchmarks, it was always maintained the same compiler optimizations so that the results could be compared to the baseline results with or without the KIR presence.

In the first code bandwidth benchmark, where CPU1 is the only one with the regulator enabled, it is shown that by regulating one of the cores, the other core, i.e., CPU0, increases in predictability. On the other hand, CPU1 is harmed in both predictability and performance. Moreover, by enabling both cores code bandwidth regulators, the results show that the lower the bandwidth given to each one of them, the best overall result will be and, therefore, the lower relative performance degradation (e.g., -21% for CPU0 and -7% for CPU1). Although KIR shows an improvement in the execution of the system by trading performance, the best result was yet to be achieved.

Next, it was studied the impact of enabling only one of the KIR features together with the regulator, i.e., *Core State* and *Synchronization*. The benchmarks for the *Core State* feature alone show that applying it without any synchronization causes, most of the time, a worse overall result for both cores.

The WCET's are achieved when synchronization is present. This result derives from the total overlap of the execution window of both CPUs. In this scenario, the performance degradation decreased only if the opposite core had less code bandwidth than the test subject being analyzed (e.g., CPU0 limited to 5000000 and CPU1 limited to 2500000).

The best overall results are achieved when both features, i.e., *Synchronization* and *Core State*, are allied to the code bandwidth regulator. Although it is possible to almost remove the existent contention, it is imperative to be careful how KIR is configured since the synchronization is enabled, which can lead to a total overlap of execution windows and portrait WCET's instead of improving the system's predictability.

Throughout the data bandwidth regulators benchmarks, it was also used the same memory layout and compiler optimizations as the one in the baseline results, i.e., shared iSRAM bank for both core's data and separate iSRAM banks for code. This benchmark aimed to verify if it could be reduced the contention existent in the iSRAMs memories when both cores use the same bank for its data placement. Although the results show a reduction in the performance degradation of the cores, it is not enough to conclude that KIR is in fact improving the predictability of the system due to the nature of the memories used, i.e., TCMs.

Lastly, the *No Contention Zone* mechanism works as expected, and the test subject, i.e., CPU0, can execute the overall benchmark without showing any performance degradation.

Therefore, it is possible to conclude that the overall KIR proposed features and mechanism when using good and aware layouts, it is possible to remove most of the existent contention related to the expansion code memory infrastructure.

5.2 uTango with KIR Benchmarking

Due to time restrictements, uTango benchmarking with KIR consists of a simple test, where each core has its own uTango image with a Coremark application ready to run on the non-secure world; thereby this benchmark is focused on verify that the mechanism is functioning as expected. Moreover, the the benchmarks uses the same memory layouts as the ones when KIR run in a bare-metal scenario. Preliminary functional tests where done to uTango to ensure that the required KIR interrupts targeted the secure world, as well that the core would enter a sleep state, if the SLEEPONEXIT bit as set, when returning from the secure to the non-secure.

Each benchmark ran without the a debugger tool; therefore, it was required modifications to the uTango and Coremark linkerfiles. Following the manual [12], the executable needs to be loaded into the FLash memory and later copied all of the code and data to their respective place. Therefore, at boot time, all of the executables are inside the (code and data) to the respective memory segment(uTango and Coremarks). At boot, each core's secure kernel, i.e., uTango, copies its data and code to the iSRAMs.

Just like the uTango, the non-secure applications of Coremark need to copy their code to eSRAMs, in case of testing the code bandwidth, and their data to the iSRAM. All of the memory regions used by the Coremark non-secure applications (i.e., QSPI Flash, eSRAM and iSRAM) neet to be configured as non-secure by uTango during configuration.

All of the Coremark non-secure applications mantain the same compiler optimization flags used when running in a bare-metal scenario. Since the optimization flags introduced floating-point instructions, it was also required that uTango enabled the floating point unit (FPU) and configure it as non-secure for this specific case. Otherwise, the non-secure application would end up in a fault.

Lastly, each one of the of Coremark applications needed to be linked with the standard library to access the input/output functions such as printf, since uTango is built without it. The Figure 5.1 allows visualization of the memory partition of the memory for two uTango executables and two non-secure applications of Coremark.

An external benchmark was required to gather the baseline results of each CPU when running Coremark without any contention or regulation. The results showed that CPU0 had a baseline result of 10.70s and CPU1 a baseline result of 39.00s. When both cores ran simultaneously, CPU0 took 19.40s to complete 1500 iterations of Coremark, i.e., -45% of performance degradation, and CPU1 took 46.07s, i.e., a performance degradation of -15%.

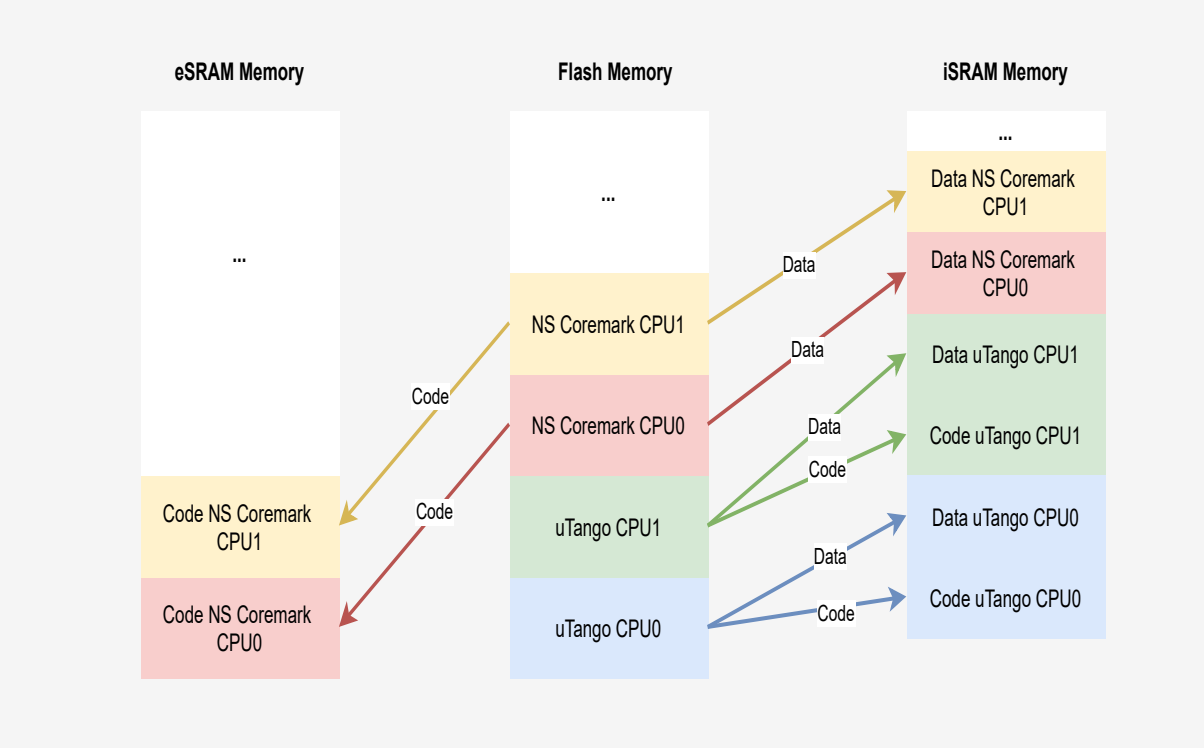


Figure 5.1: Memory layout for both cores' applications of uTango and Coremark.

5.2.1 Code Bandwidth Regulator Baseline Results

Table 5.13 depicts the Coremark baseline results for each core when running from the eSRAMs with KIR code bandwidth regulator enabled. For utangos CPU's code and data segments the internal iSRAMs were used; thereby CPU0 used the iSRAM0 and CPU1 the iSRAM2. The code segment of CPU0 non-secure Coremark is placed in the eSRAM0 and, the non-secure Coremark code segment for CPU1 is placed in the eSRAM3. The non-secure data segment of Coremark is placed in each core TCM bank, which for CPU0 is the iSRAM bank one (i.e., iSRAM1), and for CPU1 is bank three (i.e., iSRAM3). The compiler is configured with optimization flag of -O3 and the platform is set up with both CPUs running at a 50 Mhz frequency. The code bandwidth limits are maintained, i.e., 5000000, 3750000 and 2500000, as well the size of the synchronous exception (1ms) and execution window (100ms). The uTango sceduler tick was set to have a period of 10ms and each synchronization point every 1s.

					CPU0		CPU1	
CPU0		CPU1			Cache Disabled			
code	data	code	data	Iterations	Time(s)	Code Bandwidth	Time(s)	Code Bandwidth
eSRAM0	iSRAM1	eSRAM3	iSRAM3	1500	10.81	5000000	40.57	5000000
					14.20	3750000	53.28	3750000
					21.61	2500000	79.30	2500000

Table 5.13: CPU0 and CPU1 KIR Code Bandwidth Coremark baseline results.

Table 5.13 shows that, for a period of 100ms and an exception period of 1ms, the higher the bandwidth, the lower the time it will take the core to complete 1500 iterations of Coremark. Moreover, the results show that the maximum code bandwidth available in 100 ms is close to 5000000 instructions. Therefore, the time results are close to when the core executes without any regulation.

Table 5.14 depicts each core's performance degredation for the three code bandwidth limits when running solo.

CPU0			CPU1		
Time(s)	Code Bandwidth	Performance Degradation (%)	Time(s)	Code Bandwidth	Performance Degradation (%)
10.81	5000000	- 1	40.57	5000000	- 4
14.20	3750000	- 25	53.28	3750000	- 25
21.19	2500000	- 50	79.30	2500000	- 51

Table 5.14: Coremark uTango CPU0 and CPU1 base performance degredation with KIR Code Bandwidth Regulator.

5.2.2 Code Bandwidth Regulator Contention Results

Table 5.15 depicts the Coremark results obtained for each core running concurrently and fetching code from the eSRAMs. In this benchmark, both cores have the KIR code bandwidth regulators, *Core State* and *Synchronization* enabled. When CPU0 is limited to 3750000 instructions per period, the core sleeps for 25ms before starting its execution and, when limited to 2500000 instructions, it sleeps for 50ms until it starts executing.

	CPU0				CPU1			
Iterations	Time(s)	Code Bandwidth	P.D. (%)	R.P.D. (%)	Time(s)	Code Bandwidth	P.D. (%)	R.P.D. (%)
1500	19.09s	5000000	- 44	- 43	48.07	5000000	- 19	- 16
	16.34		- 35	- 34	62.12	3750000	- 37	- 14
	14.10		- 24	- 23	91.62	2500000	- 57	- 13
	25.48	3750000	- 58	- 44	45.81	5000000	- 15	- 11
	22.89		- 53	- 38	59.87	3750000	- 35	- 11
	18.58		- 42	- 24	86.76	2500000	- 55	- 8
	38.54	2500000	- 72	- 45	43.78	5000000	- 11	- 7
	32.89		- 67	- 36	56.56	3750000	- 31	- 6
	24.67		- 57	- 14	79.94	2500000	- 51	- 1

Table 5.15: Coremark contention results of uTango with KIR Code Bandwidth regulators, Synchronization and Core State enabled.

The contention results, as expected, show that by lowering the bandwidth of each core, the overall predictability increases. This is due to the presence of the *Synchronization* feature, and at the same time, the cores running in opposite phases, as long the bandwidth limit for them is inferior to 5000000 instructions. On the other hand, although the objective of the *Core State* feature is to force the opposition of execution, it is not always possible due to the size of the execution window. Therefore, overlap still occurs unless both cores have the lowest code bandwidth available, i.e., 2500000 instructions. Therefore, the contention is maximized.

For example, CPU0, when limited to 5000000 instructions, the contention is maximized when the other core also has the same amount of bandwidth. When the CPU1 has lower bandwidth, the R.P.D. of the CPU0 also decreases, but CPU1 R.P.D. stays nearly the same. Moreover, when limited to 3750000 instructions, the degradation is maximized when CPU1 has a code bandwidth of 5000000 instructions. This derives from CPU1 execution window matching the CPU0 running window. When both cores are limited to 3750000 instructions, it is created an execution window where both can execute without contention, and therefore, the decrease of both cores degradation.

Lastly, the best result is achieved when both cores are running in opposite phases, and CPU0 has an R.P.D. of -14% and CPU1 an R.P.D. of -1%. Although the degradation of CPU0 is higher than expected, it

does not mean there is contention, but on the contrary; there is the loss of performance caused by the core executing less time the expected. Hence, it takes more time to finish the same 1500 iterations. This affirmation is based on the fact that CPU1, which has not the *Core State* enabled, has a result similar to the baseline one. In case of contention, both CPU results would be higher and not just a single one.

Throughout the testing, uTango has no impact on the overall results since it uses a mapping of 1:1 in terms of applications for both CPUs and, use of good code locality. Although it is not taken into advantage the habilities of uTango, these benchmarks are used to prepare the base work for future advances in terms of increasing the predictability of the system when leading with multiple workloads.

Chapter 6

Conclusion

The last chapter of this dissertation presents the conclusions drawn from the work carried out in the Section 6.1, and some suggestions that aim to improve the work and expand the developed system (Section 6.2).

6.1 Discussion

The work carried out during this dissertation demanded the understanding of several state-of-the-art concepts and technologies. As such, several topics were studied and applied: (i) perception of contention on multi-core platforms; (ii) understanding of a new set of open-source tools and technologies (e.g., Arm TrustZone); (iii) gathering in-depth knowledge regarding the Arm CoreSight debug and tracing capabilities; (iv) comprehension of this technologies applicability focusing predictability, based on the related work.

Regarding this dissertation, it described all steps taken during the development of a system capable of acting upon possible sources of shared resources using widespread technologies in the Cortex-M architectures, make it feasible to operate across numerous platforms. The mechanism can act upon shared memories at the code and data level using the Data Watchpoint and Tracing unit functionalities by observing blocks of memory or counting the number of instructions executed by the core. With the use of a comparator and a counter of the Data Watchpoint Unit, it is possible to trigger an exception, useful to keep track of the time passed and the bandwidth used by the core. Extra functionalities such as synchronization, based on inter-core communication hardware (e.g., MHU), and core state (i.e., sleep first, execute later) lead to the further reduce the contention introduced by the used of shared resources by the cores.

In what concerns the objectives proposed at the beginning of this dissertation, the author considers that these were successfully fulfilled. The design and development of the mechanism capable of improving the system's predictability proved to be a difficult and time-consuming task. This task was composed

of the initial platform testing, which encompassed the projection of multiple memory layouts focusing on possible contention points. Moreover, the other goal's part embraced the mechanism projection and implementation using the gathered knowledge from the deep study of the debugging and tracing capabilities of the Arm debug architecture and the related work. Both stages proved to be extremely time-consuming due to the extensive set of tests and research done at various functionalities levels. On the other hand, regarding the mechanism's design and development, its implementation turned out to be challenging due to the lack and scattered documentation. To solve these issues, multiple direct contacts with Arm were required.

The integration of the produced solution in the in-house developed TEE also showed to be demanding due to the nature of the used tools and technologies the platform implemented. uTango was a single-core TEE using all open-source tools for the building and debugging of the system. Each one of the applications to be run in non-secure world needed to be compiled externally, with the concern that no memory overlaps exist, accompanied by its manual configuration in uTango. Due to the nature of uTango, every parameter needed to be defined statically by the user, ranging from the tick size of each scheduling point to the configuration of each of the resources the non-secure application would use. Any misconfiguration would lead the program to a fault. Changes in the uTango build system were also required in conjunction with a reconfiguration of some of its layers so that the system booted in multi-core scenario. Due to the presented solution's modularity,i.e., KIR, the integration time was reduced, allowing the creation of simple benchmarks focusing on the system's verification. Nevertheless, modification of the scheduler and essential structures of the TEE, and the mechanism were also needed.

The benchmarking started by gathering baseline results of each of the cores(i.e., CPU0 and CPU1) running solo, under different memory layouts, i.e., QSPI Flash, eSRAM, iSRAM. Hereafter, contention benchmarks were done again, with both cores running at the same time. The contention benchmarks consisted of testing the code and data memories part of an application, i.e., verify if there was contention related to code or data only. Therefore, the contention benchmarks only shared either data or code at a time. The code benchmarks showed that the expansion code memories, i.e., QSPI Flash and eSRAMs, have contention even if using different banks or different memories (e.g., when CPU0 uses eSRAM and CPU1 uses QSPI Flash). Moreover, when both cores use different iSRAMs for code and data, there is no contention in the system. Furthermore, the data results showed a negligible performance degradation of the cores due to the nature of the memory, i.e., tightly coupled to the core. Lastly, when used good code locality by enabling the instruction cache, the results showed a decrease in the performance degradation

of both cores.

Later, when using testing KIR, the same layouts of memory were used in conjunction with different configurations of the mechanism, i.e., bandwidth limits and state of the KIR *Core State* and *Synchronization* features, the results obtained in relation to the mechanism running in a bare-metal configuration proved to be very positive and promising regarding the experimental results.

The degradation of each of one of the cores, when using a good KIR configuration, showed to be infiror to when the core is running freely in a multi-core scenario. Moreover, the KIR *No Contention Zone* mechanism showed to be ideal for the execution of critical pieces of code since the core executes in a scenario that is equal to when running solo. Concerning the data bandwidth feature, it is considered that an alternative platform where the memories used for the data location are forcedly shared and not tightly coupled to either of the cores would create better and more visible results.

Concerning the benchmarks when uTango was used, the results, albeit having a deviation between the metric results of CPU0 and the contention results, show that the system is still viable. Regarding the mechanism feature focusing on creating zones of no contention, results matched the expected metrics. In terms of data bandwidth, it was only produced only functional tests to verify the integration is fully working and no more tests due to the results shown when running bare-metal and the nature of the iSRAM memories.

In conclusion, it is considered that the system and the architecture developed is a viable solution that corresponds in a reliable way to reduce the overal system contention and with considerably good and promising results.

6.2 Future Work

Although this dissertation's goals have been achieved, the author considers that after acquiring in-depth knowledge regarding the implemented systems and the available tools from the existing architectures, there are indeed some points where it can be improved and extended with other functionalities.

Regarding the mechanism without the integration in uTago, the suggestions are the following:

Run-time code relocation: The first suggestion consists of manipulating the code placement during run-time; thereby, critical code would be moved to a memory where no contention exists (e.g., an iSRAM bank). This could be achieved by modifications of the linkerfile and the compiling flags of that piece of code. Therefore, KIR would be responsible for the loading and cleaning of the used memory blocks.

Cache manipulation: The second suggestion consists in the use of cache techniques to trap critical code in the cache, avoiding their eviction, as shown in literature [93]. This way, performance would increase together with predictability. To achieve this goal, the platform must support a mechanism capable of manipulating the its cache lines.

Use a tracing buffer: The third suggestion requires the existence of extra tracing hardware, such as an Embedded Trace Buffer where packets coming from tracing sources could be stored. Although the packets would be highly compacted, the use of available hardware filtering functions, it could be possible to minimize the invasive debugging used previously. Therefore, the core could trace its applications execution without the necessity of invasive techniques.

Regarding uTango integration, there are still points where the mechanism could be improved. Therefore the following suggestions are regarding what could be added.

Implement a global bandwidth manager: The first suggestion would be to create a global bandwidth manager that could collect bandwidth not used by the core. Each core would have a global budget that could be distributed between its applications, and, in case it was not all used, it could be donated so that the other core could claim it. A predictor could be implemented so that, in run-time, the core could know when it was expected some part of the budget to not be used.

Change uTango scheduling points: The second sugestion, is to force a scheduling point when the application deplets its available bandwidth. If the core, still had available, bandwidth a new world is scheduled and runs until it deplets its or the core bandwidth. Hence, another application would start to execute immediately, and the one that runs out of bandwidth could not be scheduled again until a new period starts. Thus, dead periods could be minimized, increasing the performance and predictability of the system.

Lastly, the last suggestion is not related to increase or changing the of KIR, but instead the repetition of the data bandwidth regulator benchmarks in a platform where the memory used for the data location is of a different nature. The aim is to conclude if the KIR data bandwidth regulator is, in fact, effective or not.

Appendices

```
#ifdef CORE_0

kir_core kir_core_config =
{
    .recharge_period = 100, /* 100ms */
    .exception_period = 1, /* 1ms */
    .kir_sync = (kir_sync[])
    {
        {
        .sync_period = 1000, /* 1s */
        .core_coms = (kir_core_coms[])
        {
                {
                .sent_message = MessageClear,
                .received_message = MessageClear,
                .mhu = MHU0,
                .mhu_dev = &ARM_MHU0_DEV_S,
                .cpu_id = ARM_MHU_CPU1,
                }
        },
        .Timer = CMSDKTIMER1,
        .status = on,
        }
    },
    .core_state = (kir_core_state[])
    {
            {
                .sleep_time = 50,
                .status = off,
            }
    },
    .mem_band = (kir_mem_band[])
        {
            {
                .max_mem_access_count = 1000,
                .mem_watch_start = 0x3000012c,
                .mem_watch_size = 0x100,
```

```
38                  .status = off,
39              }
40          },
41          .code_band = (kir_code_band[])
42          {
43              {
44                  .max_instruction_count = 2500000,
45                  .status = on,
46              }
47          }
48      .status = on,
49 };
50 #endif
```

Listing 1: CPU0 KIR Global Instance Configuration Example.

Bibliography

[1] Arm, "Introduction to the ARMv8-M architecture," 2017.

[2] J. Andrews, "Using DS-5 Trace with Arm Fast Models," 2017. [Online]. Available: https://community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/using-ds-5-trace-with-arm-fast-models

[3] Arm, "ARMv8-M Architecture Reference Manual," 2017.

[4] J. Yiu, "ARMv8-M Architecture Technical Overview," 2015.

[5] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted Execution Environment: What It is, and What It is Not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, aug 2015, pp. 57–64.

[6] D. Oliveira, T. Gomes, and S. Pinto, "uTango: an open-source TEE for the Internet of Things," pp. 1–14, 2021.

[7] M. Xu, L. Thi, X. Phan, H. Choi, and I. Lee, "vCAT: Dynamic Cache Management Using CAT Virtualization," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, apr 2017, pp. 211–222.

[8] P. Modica, A. Biondi, G. Buttazzo, and A. Patel, "Supporting temporal and spatial isolation in a hypervisor for ARM multicore platforms," in *2018 IEEE International Conference on Industrial Technology (ICIT)*, feb 2018, pp. 1651–1657.

[9] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 562–576, feb 2016.

[10] A. Bansal, R. Tabish, R. Mancuso, R. Pellizzoni, and M. Caccamo, "Evaluating the Memory Subsystem of a Configurable Heterogeneous MPSoC," *Proceedings of the 14th annual workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT)*, 2018.

[11] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares, "Virtualization on TrustZone-Enabled Microcontrollers? Voilà!" in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, apr 2019, pp. 293–304.

[12] Arm, "Arm Musca-A Test Chip and Board Technical Reference Manual," January 2018.

[13] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context Aware Computing for The Internet of Things: A Survey," *IEEE Communications Surveys Tutorials*, vol. 16, no. 1, pp. 414–454, 2014.

[14] P. Sparks, "The Route to a Trillion Devices," Arm, Tech. Rep. June, 2017. [Online]. Available: https://community.arm.com/iot/b/internet-of-things/posts/white-paper-the-route-to-a-trillion-devices

[15] C. B. Watkins and R. Walter, "Transitioning from federated avionics architectures to Integrated Modular Avionics," in *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*, oct 2007, pp. 2.A.1–1–2.A.1–10.

[16] F. Bruns, D. Kuschnerus, and A. Bilgic, "Virtualization for Safety-Critical, Deeply-Embedded Devices," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 1485–1492.

[17] M. Paulitsch, O. M. Duarte, H. Karray, K. Mueller, D. Muench, and J. Nowotsch, "Mixed-Criticality Embedded Systems – A Balance Ensuring Partitioning and Performance," in *2015 Euromicro Conference on Digital System Design*, aug 2015, pp. 453–461.

[18] S. Baruah, H. Li, and L. Stougie, "Towards the Design of Certifiable Mixed-criticality Systems," in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, apr 2010, pp. 13–22.

[19] Intel, "The Benefits of Virtualization for Embedded Systems," Tech. Rep.

[20] H. Kim and R. R. Rajkumar, "Predictable Shared Cache Management for Multi-Core Real-Time Virtualization," *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 1, dec 2017.

[21] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems," 2020.

[22] Arm, "ARMv6-M Architecture Reference Manual," Arm, Tech. Rep., 2010.

[23] Arm, "ARMv7-M Architecture Reference Manual," Arm, Tech. Rep., 2014.

[24] Arm, "Arm TrustZone Technology for the Armv8-M Architecture," Tech. Rep., 2016.

[25] R. Pan, G. Peach, Y. Ren, and G. Parmer, "Predictable Virtualization on Memory Protection Unit-Based Microcontrollers," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, apr 2018, pp. 62–74.

[26] S. Pinto, A. Oliveira, J. Pereira, J. Cabral, J. Monteiro, and A. Tavares, "Lightweight multicore virtualization architecture exploiting ARM TrustZone," 2017, pp. 3562–3567.

[27] S. Pinto, J. Pereira, T. Gomes, M. Ekpanyapong, and A. Tavares, "Towards a TrustZone-Assisted Hypervisor for Real-Time Embedded Systems," *IEEE Computer Architecture Letters*, vol. PP, 2016.

[28] S. Pinto, T. Gomes, J. Pereira, J. Cabral, and A. Tavares, "IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices," *IEEE Internet Computing*, vol. 21, no. 1, pp. 40–47, jan 2017.

[29] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 67–80.

[30] S. Pinto, D. Oliveira, J. Pereira, J. Cabral, and A. Tavares, "FreeTEE: When real-time and security meet," 2015.

[31] A. Sadeghi, C. Wachsmann, and M. Waidner, "Security and privacy challenges in industrial Internet of Things," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, jun 2015, pp. 1–6.

[32] R. Ernst and M. Di Natale, "Mixed Criticality Systems—A History of Misconceptions?" *IEEE Design Test*, vol. 33, no. 5, pp. 65–74, oct 2016.

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

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

[35] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A predictable SDRAM memory controller," in *2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007, pp. 251–256.

[36] I. Insights, "Research Bulletin," *Library*, no. August 25, 2020. [Online]. Available: https://www.icinsights.com/news/bulletins/MCUs-Expected-To-Make-Modest-Comeback-After-2020-Drop--/

[37] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens, "Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration," in *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2008, pp. 3–14.

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

[39] M. Caccamo, G. C. Buttazzo, and D. C. Thomas, "Efficient reclaiming in reservation-based real-time systems with variable execution times," *IEEE Transactions on Computers*, vol. 54, no. 2, pp. 198–213, feb 2005.

[40] C. Berg, J. Engblom, and R. Wilhelm, "Requirements for and Design of a Processor with Predictable Timing," 2004.

[41] H. Wang, N. C. Audsley, and W. Chang, "Addressing Resource Contention and Timing Predictability for Multi-Core Architectures with Shared Memory Interconnects," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 70–81.

[42] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-Aware Scheduling and Analysis for Multicores," in *Proceedings of the Seventh ACM International Conference on Embedded Software*, ser. EMSOFT '09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 245–254.

[43] J. Yan and W. Zhang, "WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches," in *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, 2008, pp. 80–89.

[44] J. Rosen, A. Andrei, P. Eles, and Z. Peng, "Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip," in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, 2007, pp. 49–60.

[45] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O," in *2012 IEEE International Conference on Cluster Computing*, 2012, pp. 155–163.

[46] J. Yiu, "Software Development in ARMv8-M Architecture," in *Proceedings of Embedded World*, 2017.

[47] J. Yiu, "ARM Cortex-M for Beginners; An overview of the ARM Cortex-M processor family and comparison," *ARM Limited, White Paper*, no. September, pp. 1–25, 2016.

[48] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A Comprehensive Survey," *ACM Comput. Surv.*, vol. 51, no. 6, jan 2019.

[49] S. Pinto and C. Garlati, "Multi Zone Security for Arm Cortex-M Devices," 2020.

[50] Arm, "ARMv8-M Exception Connect User Guide handling," Tech. Rep., 2016.

[51] Arm, "ARMv8-M processor power management," Tech. Rep., 2016.

[52] J. Yiu, *The Definitive Guide To Arm Cortex-M3 and Cortex-M4 Processors*, 2013.

[53] Arm, "ARMv8-M Processor Debug," Tech. Rep., 2016.

[54] Arm, "CoreSight Technical Introduction A quickstart for designers," Tech. Rep. August, 2013.

[55] Arm, "CoreSight Components," Tech. Rep., 2009.

[56] Arm, "ARM CoreSight SoC-400," Tech. Rep., 2009.

[57] Arm, "ARM Embedded Trace Macrocell Architecture Specification," Tech. Rep., 2013.

[58] Arm, "Understanding Trace," Tech. Rep.

[59] J. Andrews, "Trace Cortex-M software with the Instrumentation Trace Macrocell (ITM)." [Online]. Available: https://community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/trace-cortex-m-software-with-the-instruction-trace-macrocell-itm

[60] Arm, “Embedded Trace Buffer,” Tech. Rep., 2002.

[61] Arm, “ARM Security Technology Building a Secure System using TrustZone Technology,” Tech. Rep., 2009.

[62] Qualcomm, “Guard your data with the Qualcomm Snapdragon Mobile Platform,” Tech. Rep., 2019.

[63] SierraWare, “SierraTEE Trusted Execution Environment.” [Online]. Available: https://www.sierraware.com/open-source-ARM-TrustZone.html

[64] Arm, “Arm Trusted Firmware.” [Online]. Available: https://www.trustedfirmware.org/

[65] Trustonic, “Trustonic Kinibi-M.” [Online]. Available: https://www.trustonic.com/technology/

[66] Prove&Run, “ProvenCore-M.” [Online]. Available: https://www.provenrun.com/products/provencore/

[67] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, “SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems,” *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2020-May, pp. 1416–1432, 2020.

[68] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. V. Hanxleden, R. Wilhelm, and W. Yi, “Building Timing Predictable Embedded Systems,” *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4, mar 2014.

[69] J. Freitag, S. Uhrig, and T. Ungerer, “Virtual Timing Isolation for Mixed-Criticality Systems,” in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, 2018.

[70] A. Nogueira and M. Calha, “Predictability and Efficiency in Contemporary Hard RTOS for Multiprocessor Systems,” in *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, vol. 2, 2011, pp. 3–8.

[71] Siemens, “Jailhouse.” [Online]. Available: https://github.com/siemens/jailhouse

[72] X. Project, “Xen,” p. 6. [Online]. Available: https://xenproject.org/users/virtualization/

[73] L. Project, “LITMUS,” p. 634. [Online]. Available: https://www.litmus-rt.org/

[74] Intel, "Intel Performance Counter Monitor - A Better Way to Measure CPU Utilization," 2012. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-performance-counter-monitor.html

[75] Arm, "Using the PMU Event Counters in DS-5." [Online]. Available: https://developer.arm.com/tools-and-software/embedded/legacy-tools/ds-5-development-studio/resources/tutorials/using-the-pmu-event-counters-in-ds-5

[76] X. Project, "XVISOR," p. 6. [Online]. Available: http://xhypervisor.org/

[77] Heechul Yun, "IsolBench." [Online]. Available: https://github.com/CSL-KU/IsolBench

[78] Standard Performance Evaluation Corporation, "SPEC CPU 2006." [Online]. Available: https://www.spec.org/cpu2006/

[79] R. Lab, "Erika Enterprise RTOS v3," p. 6. [Online]. Available: http://www.erika-enterprise.com/

[80] Arm, "Arm CoreLink SSE-200 Subsystem for Embedded," Tech. Rep., 2018.

[81] Arm, "ARM Cortex-M33 Processor," Tech. Rep., 2017.

[82] Arm, "GNU Arm Embedded Toolchain." [Online]. Available: https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm

[83] GNU Software, "GNU Make." [Online]. Available: https://www.gnu.org/software/make/

[84] Pyocd, "pyOCD." [Online]. Available: https://github.com/pyocd/pyOCD

[85] E. Foundation, "GNU MCU Eclipse." [Online]. Available: https://eclipse-embed-cdt.github.io/

[86] John D. McCalpin, "STREAM Benchmark." [Online]. Available: https://www.cs.virginia.edu/stream/

[87] EEMBC, "AutoBench," 2554. [Online]. Available: https://www.eembc.org/autobench/

[88] EEMBC, "MultiBench." [Online]. Available: https://www.eembc.org/multibench/{#}:{~}:text=MultiBench™isasuite,supportforfinegrainparallelism.

[89] S. Gal-on and M. Levy, "Exploring CoreMark™ - A Benchmark Maximizing Simplicity and Efficacy," *The Embedded Microprocessor Benchmark Consortium (EEMBC)*, 2012.

[90] Arm, “CoreMark Benchmarking for ARM Cortex Processors,” Tech. Rep., 2013.

[91] Arm, “Using DWT and other methods to count executed instructions on Cortex-M.” [Online]. Available: https://developer.arm.com/documentation/ka001499/1-0/

[92] D. Butcher, “How to Call a Function from Arm Assembler,” 2013. [Online]. Available: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/how-to-call-a-function-from-arm-assembler

[93] Microchip, “How to Achieve Deterministic Code Performance Using a Cortex-M Cache Controller,” Tech. Rep., 2018.