UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DISCA
DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

Máster en Ingeniería de Computadores y Redes
Trabajo Fin de Máster

Measuring and Controlling Multicore Contention in a RISC-V System-on-Chip

Author: Pablo Andreu Cerezo

Tutor: Pedro Juan López Rodríguez

External Tutor: Carles Hernandez Luz

Course 2020-2021

Resum

Els processadors multinucli van començar una revolució en el còmput modern quan van ser introduïts en l'espai de còmput comercial i de consumidor. Aquests processadors multinucli presentaven un augment significatiu en consum, eficiència i rendiment en un període de temps en l'augment de la freqüència i l'IPC de l'processador semblava estar tocant sostre. No obstant això, en sistemes crítics, la introducció dels processadors multinucli ha portat a la llum diferents dificultats en el procés de certificació. La principal àrea que dificulta la caracterització dels sistemes multinucli en temps real és l'ús de recursos compartits, en específic, els busos compartits.

En aquest treball proveirem les eines necessàries per facilitar la caracterització de sistemes que fan ús de busos compartits en sistemes de criticitat mixta. En específic, combinem les polítiques desenvolupades per a sistemes amb busos amb polítiques de limitació d'ample de banda basades en interferència causada a el nucli principal. Amb aquesta combinació de polítiques podem limitar l'WCET de la tasca crítica en el sistema multinucli mentre que proveïm un "best effort"per permetre el progrés en els nuclis secundaris.

Paraules clau: Criticalitat mixta, Bus AHB, Arbitratje en multicore, Quota, Sistema en temps real

Resumen

Los procesadores multinúcleo empezaron una revolución en el cómputo moderno cuando fueron introducidos en el espacio de cómputo comercial y de consumidor. Estos procesadores multinúcleo presentaban un aumento significativo en consumo, eficiencia y rendimiento en un periodo de tiempo en el aumento de la frecuencia y el IPC del procesador parecía estar tocando techo. Sin embargo, en sistemas críticos, la introducción de los procesadores multinúcleo ha traído a la luz diferentes dificultades en el proceso de certificación. La principal área que dificulta la caracterización de los sistemas multicore en tiempo real es el uso de recursos compartidos, en específico, los buses compartidos.

En este trabajo proveeremos las herramientas necesarias para facilitar la caracterización de sistemas que hacen uso de buses compartidos en sistemas de criticidad mixta. En específico, combinamos las políticas desarrolladas para sistemas con buses con políticas de limitación de ancho de banda basadas en interferencia causada al núcleo principal. Con esta combinación de políticas podemos limitar el WCET de la tarea crítica en el sistema multinúcleo mientras que proveemos un "best effort" para permitir el progreso en los núcleos secundarios.

Palabras clave: Criticalidad mixta, Bus AHB, Arbitraje en multicore, Quota, Sistema de tiempo real

Abstract

Multicore processors were a revolution when introduced into the commercial computing space, they presented great power efficiency and performance in a time where clock speeds and instruction level parallelism were plateauing. But, on safety critical systems, the introduction of multi-core processors has brought serious difficulties to the certification process. The main trouble spot for multicore characterization is the usage of shared resources, in specific, shared buses.

In this work, we provide tools to ease the characterization of shared bus mechanisms timing interference on critical and mixed criticality systems. In particular, we combine

shared bus arbitration policies with rate limiting policies based on critical workload interference to bound the WCET of a critical workload on a multi-core system while doing a best effort to let secondary cores progress as much as possible.

Key words: Mixed Criticality, AHB bus, Multicore arbitration, Quota, Real-time system

Contents

List of Figures

List of Tables

CHAPTER 1
Introduction

In current times, the electronic systems used on the industry, automotive, avionics and consumer transport need to warrant extremely high safety levels. This high safety levels currently are determined following several certification standards that vary depending on the application and industry. For instance, in the context of automotive electronic products components need to adhere to the ISO26262 [1].

With the need for more computing for increasingly complex tasks, like the ones executed in an autonomous car, multicore and high performance processors are needed. Unfortunately, the certification of multicore processor including aggressive AI (Artificial Intelligence) accelerators greatly hinders the ability to certify modern systems with the safety levels needed for certain applications. As an example, fully autonomous cars require the computing elements to be certified according to the highest integrity levels (ASIL-D according to ISO26262 nomenclature).

Multicore processors with many shared resources increase the complexity of the timing verification step imposed by functional safety certification standards. The higher the criticality the higher the evidence that must be collected to ensure timing budgets assigned to specific tasks are met. The main reason why multicore processors increase the complexity of the verification process relies on the difficulties to accurately and safely quantify the interference that can occur due to interactions with other co-running tasks in the different shared resources.

In this work, our goal is to ease the certification and verification process of high-performance multicore processors and in particular the timing verification step. For that purpose, we propose several techniques aimed at improving the policies that control the access to the shared resources for tasks that are subject to different criticality levels (mixed-criticality).

This work is implemented as a part of the SELENE H2020 European project. This means that the effect of the implemented policies on the real world will be high. In this project, we are using a four-core computing platform that targets the execution of mixed-criticality applications. In this work, we consider mixed-criticality applications in which a highly-critical application is co-running with other non highly-critical applications. Thus, we prioritize a critical core over three non-critical or less critical cores while trying to guarantee progress on these non-critical cores bounding the interference that the critical core can suffer from this non-critical worlds. This four core real time system is the SELENE SoC (System on Chip), where we have four dual issue in-order cores sharing a bus to memory. This bus implements an AMBA AHB on-chip bus to handle the access to the memory hierarchy. AMBA is the most common standard for high performance shared buses. This means that the results of this work can easily be ported to any other system that implement the AMBA AHB standard.

The focus of the developed policies will be to mitigate the interference between cores on the multiprocessor system and to be able to bound a WCET (Worst Case Execution Time) for tasks running on different cores. Thus, bounding the interference between these cores.

1.1 Motivation

The usage of shared buses is both prevalent and necessary on modern microprocessors. But, with the usage of this shared buses comes the problem of bounding the potential interference between different tasks running on different cores and sharing resources, such as the AHB bus and the memory subsystem.

The traditional approach to multicore systems on certifiable real time systems has been to disable the concurrent usage of multiple cores and completely partition all resources, either by hardware or software means, but this approach leads to a great amount of hardware redundancy and thus, unused resources.

With this work we aim to provide a platform where multiple criticality applications can be running on multiple cores and still remain certifiable, greatly improving the SWaP (Size Weight and Power consumption) of the final system.

The improvement on SWaP is due to sharing already available resources between tasks that commonly would be ran on an individual chip each, thus, making an overall more efficient system. This effort to simplify previously complex systems is being developed on all industries, from avionics [2] where there is a high degree of redundancy and the workloads ran are very critical, to automotive [3], where the cost savings of having less components is significant.

The mixed-criticality approach for soft real time systems would lead to a reduction on overall chips needed due to the work of multiple chips being carried by this one chip, improving resource utilization and greatly reducing supply chain constraints.

This supply chain constraints become specially obvious on the automotive industry on recent times, where, with the ongoing chip shortage of 2021, some automotive supply chains needed to be stopped due to the shortage of one computer component out of many. With multiple use processors that leverage the technology presented on this work, the probability of chip shortage supply chain constraints gets reduced greatly by aggregating multiple critical functionality on the same chip, and thus, reducing the number of needed components and the probability of supply chain failure.

This approach reduces supply chain constraints, total cost of the system, probability of failure of the system, radiation exposure (as having more space efficient computing reduces die area exposed to radiation) and power consumption. However, this comes at the cost of increasing the complexity of certification. With this work we aim to close this gap by by offering hardware support for improving the predictability of mixed-criticality tasks running on the same SoC.

1.2 Objectives

The objectives of this work are the following:

- Analyse the contention caused to the main workload from the other cores on the AHB shared resource.

- Bound the contention caused to the most critical workload using low-cost hardware policies.
- Improve the computing capabilities of the system without comprising the timing verification of critical tasks
- Reduce the impact of pathological cases such as the synchronization effect described on [4, Section 4].

With those objectives in mind, we thoroughly analysed the different options and objectives of the SELENE H2020 project (where this work is developed) and decided to implement some secondary objectives:

- Extend the already implemented BSC (Barcelona Supercomputing Center) PMU (Performance Monitoring Unit) functionality to obtain new performance metrics.
- Use the BSC PMU quota mechanism as a baseline to implement a new family of shared resources arbitration mechanisms.
- Implement software configuration for the implemented functionality.
- Use the Embassy Coremark® PRO benchmark suite to test our policies.
- Test the hardware policies on synthesized hardware as to have a comprehensive set of results.
- Test the results on the synthesized SELENE SoC hardware.
- Compare the results with the baseline SELENE AHB policies.
- The design must be able to be easily configured and expanded for more complex capabilities.

1.3 Expected impact

The expected impact of the policies and mechanisms presented in this work is substantial. The modern embedded critical system paradigm is becoming more complex every day, and contemporary problems require contemporary solutions.

With mixed-criticality systems, much of the cars subsystems can be implemented on the same computing module with shared resources, thus, greatly improving supply chain efficiency and being able to implement more features on less silicon.

Hardware support for this mixed-criticality systems is at the core of the software hypervisor support and certification process, as the certifier would be able to threat each independent application running on the core as it was running standalone. Thus, being able to calculate their expected WCET (Worst Case Execution Time) in a confident manner.

Indeed, as this work is part of the SELENE H2020 project, its deployment in a realistic environment is guaranteed, having thus a high expected impact on the industrial world. It also would enable the grouping of different tasks and types of tasks on the same microchip. Also reducing its energy consumption, the radiation exposed area (in the case of a satellite) and total cost.

Indeed, greatly improving the space efficiency, energy and physical resource utilization without giving up the capacity to execute critical tasks with predefined upper bounds on shared resources.

1.4 Methodology

As for the methodology, we have used the SELENE SoC as a baseline to implement our proposed hardware AHB policies. The first step has been the characterization of the performance of this platform. Later, we have investigated the inner workings of the PMU (Performance Monitoring Unit) as well as the problems that it uncovered on its test on the SELENE SoC [4].

We then aimed to solve the limitations of the existing policies on the the shared bus that the PMU unveiled. From intra-task synchronism to bounding inter-core interference on the AHB bus. Thus, expanding the PMU original functionality to adapt it to our needs.

After the initial analysis, we decided to implement several policies by modifying the existing SoC RTL. The SoC RTL is written in vhdl and to implement our different policies we have mainly modified the AMBA AHB controller from Cobham Gaisler.

Finally, we tested our modifications first on simulation, using Questasim, and later on the FPGA prototype by synthesizing the design and running extensive tests. From the FPGA and using the GRMON tool, we extracted the results that will be later presented on this work.

1.5 Memory structure

This document is structured as follows In chapter 2 some background is presented to understand the context and importance of this work. We define the importance of mixed criticality on embedded systems, the AMBA AHB standard, that is the standard that our interconnection network uses. We also present how WCET calculus is done on critical systems and some arbitration policies that exist for critical systems and shared buses, as well as some common arbitration policies.

In chapter 3 we define the system where this work is developed. From its components to its cores, memory subsystem and interconnection network.

In chapter 4 we present our proposed changes to the system to improve performance and bound contention.

In chapter 5 we present the effect that the proposed changes have on our system performance.

In chapter 6 we end this work by summarizing the most important changes made to the system and its effect on performance guarantees and overall efficiency.

Finally, on appendix A we present the tools used to develop and deploy this work and this works relation with the master degree studied.

CHAPTER 2
Background

This chapter summarizes the state of the art on multicore contention analysis and control in real-time embedded systems. First, we introduce the concept of a mixed criticality system. Then introducing WCET (Worst Case Execution Time) calculus both in static and probabilistic manner. After WCET calculus is presented, different mechanisms of preserving WCET on shared bus systems are presented. From the traditional TDM(Time Division Multiplexing) to more innovative solutions.

Then we proceed to explain the effect that multicore arbitration policies for shared buses have on the final system. By finally presenting the AMBA AHB standard. This standard is critical to our work as is the most common shared bus standard and the one that our system uses, so it will determine the characteristics of our final implemented solution.

2.1 Mixed criticality on embedded systems

In embedded systems certification of certain safety standards is needed. This functional safety standards define several criticality levels that are associated to the different functionalities of a given system. For instance, using the Automotive nomenclature defined in the ISO2622 [1] ASIL-D corresponds to the highest criticality level while QM is the term used to define those functionalities that are not relevant for the safety of the system and just need to be quality managed (QM). Regardless of the domain of application, the higher the criticality level the higher the consequences that can occur after a failure in such functionality.

With the advent of multicore processors the integration of several functionalities of different criticality (a.k.a mixed-criticality applications) has been considered a very interesting approach to reduce the size, weight, area and power costs (SWAP) of critical electronic equipment. However, while mixed-criticality integrated systems have obvious advantages they also introduce new complexities to the verification and validation process imposed by functional safety certification standards.

In a mixed-criticality system, the easiest way to certify an application is to isolate its functioning from other components by physically having different chips for different functionalities. A simple example of this traditional approach is having a component for controlling the ABS (Anti-lock braking system), one for the ESP (Electronic stability control) of a car and another component for controlling the infotainment screen.

This physical hardware separation combined with static scheduling policies greatly eases the certification of the system. But with this ease of certification comes the burden

of power consumption, size and weight of the system. So this simplicity comes at the cost of SWaP[5].

As said before, with the advent of modern multicore processors, sharing functionality between subsystems with different criticalities on the same processor to fully exploit its computing power is being discussed. In the context of multicores, this requires implementing hypervisors and virtual machines to isolate the critical worlds between each other both in spatial and temporal domains [5], but tight control of shared resources is needed to bound the WCET (Worst Case Execution Time).

This multiprocessor paradigm and the control of shared resources requires a great deal of optimizations to prevent or bound the interference of lower criticality tasks to higher criticality tasks on shared resources. This tight control of shared resources will let us take the most possible performance out of our system without sacrificing time predictability.

In order to accomplish this, control mechanisms need to be put in place on shared resources to bound the worst case execution time. This can be done on a deterministic manner by calculating the WCET of an application on the system, or in a probabilistic way, by calculating the probabilistic WCET of this system. The probabilistic approach greatly simplifies the calculation of WCET of the studied task but it also requires the system to meet certain properties.

"To break the dependence between tasks, we design our multicore such that we make that the worst effect that one task can incur on the execution of any other task due to inter-task interferences can be probabilistically bounded. [6, PTA in multicore systems]"

To obtain this task independence we implemented several policies on the shared AHB bus to mitigate and bound the effect of non-critical tasks on the critical world. Those policies will be further discussed on chapter 4.

2.2 WCET estimation

WCET (Worst Case Execution Time) calculus is the act of analyzing a task and the task running on a given system with its operating system, memory subsystem, cache subsystem, branch prediction, processor pipelines and the internal processor structure itself and predicting the worst possible execution time for it based on some criteria.

2.2.1. Deterministic WCET

For deterministic WCET the goal is to determine the worst case execution time for a task on a given processor. This can be done with static analysis and measurement-based techniques.

With the static analysis approach, a study of the timing model of the hardware and the software is performed and a prediction of the upper bound on the longest path of the program is made. This prediction is made using tree based approaches [7] and path based approaches [8]. But this prediction methods become unfeasible as the system becomes complex.

For a system like the SELENE platform, that uses a NOEL-V system and Linux as Operating System, static analysis is extremely complex and resource intensive. Additionally, this often leads to overly pessimistic worst cases but guaranteeing that this worst case is never reached. However. at the same time, this approach can result on oversights of the

real system that fail to take into account for exceptional phenomena that can result on a real run.[9]

An alternative to estimating WCET using the previously described static analysis is to use analysis by measurement. With this approach the code is run under exhaustive test conditions and the longest execution time is recorded. This measurement practice has the advantage that uses the real system and is cost-effective, but can fail to determine the worst possible case due to the lack of knowledge of all potential situations, and thus, not estimate the worst possible execution time.

To accommodate this uncertainty, some margin of safety is usually added to the performed calculations to absorb any possible not contemplated case. The usage and value of this margin is not scientifically determined and is usually based on the experience of the test engineers. A high safety margin results in performance losses while a low safety margin can lead to violations of the WCET[9]. Unfortunately, as the complexity of the system increases the confidence on the established safety margin decreases significantly.

2.2.2. Probabilistic WCET

> The aim of probabilistic WCET analysis is to determine the probability distribution of the worst-case execution time of a particular code fragment. The problem is formulated (and solved) in terms of a syntax tree representation of the program and a probabilistic timing schema.[9]

To construct this probabilistic timing schema successfully, the probability of missing a deadline with pWCET should be the same as the probability of missing a deadline with other dependability estimates. To provide this probability estimates for pWCET, determining the probability distribution of the execution time of individual tasks is necessary.

With pWCET it is possible to combine the pros of static WCET calculus and measurement WCET calculus. It uses the execution times of individual blocks observing the real-system and leverages static analysis techniques to obtain the worst case effects of the processor and task under analysis. However, the most commonly used approach in the context of probabilistic analysis is the utilization of timing measurements. This approach is often now as measurement-based probabilistic analysis (MBPTA).

With MBPTA a program is executed a given number of times and timing measurements are collected to form a probability distribution. Finally, to be able to determine arbitrarily low timing violation probabilities extreme value theory is used.

2.3 Shared Resources Arbitration

We find two main approaches to deal with contention in shared resources. The first approach is removing the contention by construction using control access protocols to handle the access to the shared resource. The other approach, and the one we also follow in this work, is using common or general purpose arbitration policies and use quotas to avoid tasks creating uncontrolled interference.

2.3.1. Time Division Multiplexing

Traditionally, interconnect solutions for real-time systems with strict deadlines were based on the utilization of time-division multiplexing (TDM). TDM avoids having conflicts in

shared resources by assigning CPU cores exclusive access to resources for certain time slots.

Examples of TDM multicore systems using a bus have been proposed in [10] and [11] The utilization of TDM in NoCs is more difficult due to the distributed nature of the arbitration. For instance, T-CREST and DTU NoCs use an offline scheduling process to find conflict free transmissions and this offline process is only able to find suitable schedules with relatively small periods for small NoC configurations. A more recent and effective approach in the one proposed by DCFNoC [12]. DCFNoC uses the channel dependency to injects delays in the critical points and allow conflict-free arbitration.

Approaches based on TDM simplify the derivation of WCET in multicore systems by providing a constant transmission time for requests using the interconnect. However, these solutions are not well suited to applications requiring real-time high computational power demands since they have a significant impact in the best effort performance of the applications executed in these platforms.

2.3.2. Contention on priority based shared bus systems

Shared bus systems are incredibly efficient on using the available shared resources such as memory. But contention to the main core workload from other bus entities needs to be properly managed in order to not violate WCET estimates.

Shared bus systems increase the entropy of a system and can lead to pathological cases. This can manifest as traffic patterns on the shared resources that lead to pathological contention to the critical workload.

To ease and bound the effect of this pathological cases some systems have been researched such as MemGuard [13]. MemGuard divides the available memory bandwidth on two categories, guaranteed and best effort.

Due to the characteristics of DRAM memory the available memory bandwidth can vary depending on its access pattern, and this access pattern can be difficult to predict on multicore systems, that is why guaranteeing a minimal progress of a workload is not as simple as giving it some percentage of the total maximum DRAM or bus throughput.

Memguard tries to deal with the varying throughput of the shared DRAM by estimating its worst possible throughput given a test-case and dividing that "guaranteed" throughput between the different memory contenders. This prediction of minimal throughput can be wrong, leading to worse than expected performance, and thus, missed deadlines. But, if no miss-characterization of access patterns is produced, this "guaranteed" throughput will be met and more than worst case progress can be made.

This is on contrast with TDM, where the bus arbitration is predetermined statically on system design and it is impossible to obtain better than "guaranteed" performance.

On contrast with the Memguard approach of meeting quotas on each time period is the definition of interference sensitive WCET (isWCET) presented on [14] that takes into account inter-process interference due to the use of shared resources in multi-core processors.

With this approach, instead of limiting the contention between inter process interference on a shared bus we try to calculate the worst possible case of each process independently and of the shared resource. By combining them offline, we can compute the "isWCET" for each task without having to take into account the whole system, thus, simplifying the calculus of WCET.

With the approach defined on [14] we cater to the worst possible contention on the critical workload, thus, greatly increasing our critical task computing time.

In summary, there are several approaches to bound contention on shared bus systems, from calculating the worst possible case of the shared resource and catering the expected runtime of the critical task to it [14]. To trying to guarantee some performance to several cores by handing guaranteed performance quotas to the bus actors [13] and doing a best effort to make more than guaranteed progress.

2.4 Multicore arbitration policies

In terms of multicore arbitration policies, the most common ones are Round-Robin (section 3.3) and a static priority policy (section 4.1).

With the round robin arbitration policy, a register of the last served request is stored and the next index that is requesting the bus will be served, wrapping around on a circular manner. Theoretically, for a fully fair round robin policy all bus participants should be arbitrated with the same average latency and bandwidth should be distributed fairly.

With the static arbitration policy, an offline analysis of bus participants is made and all participants are put on a priority list, from higher priority to lower priority. Then, as they request the bus, the bus token is given to the requesting master with a higher priority. This arbitration mechanism should lead to higher priority masters having a much lower average latency on their worst case and a much higher throughput, but can end up starving the low priority masters.

With the lottery bus arbitration policy described on [6, Lottery Bus] each arbitration round the arbiter selects a master of the bus randomly to give it access to the shared bus. This random access to the bus decouples the different master traffic patterns uniformizing the worst cases of bus access that we observed on the previous section. This uniformization of the worst cases by reducing pathological bad bus traffic patterns comes at the cost of best case performance of the bus.

In conclusion, there is not a clear best arbitration policy that can be used by default on a real-time multi-core system that guarantees the best possible performance. They all come with trade-offs and should be taken into account when the systems timing analysis is being performed.

2.5 AMBA AHB standard

"AMBA AHB is a bus interface suitable for high-performance synthesizable designs. It defines the interface between components, such as masters, interconnects, and slaves."[15]

The AHB (Advanced High-performance Bus) bus is a shared bus widely used in chip designs due to it's shared nature. The most common applications for AHB are internal memory devices, external memory interfaces and high bandwidth peripherals. Even though it is a shared bus it has high performance characteristics such as burst transfers, cache coherency possibilities, split transactions and bus locking mechanisms. It has a lower performance counterpart called APB (Advanced Peripheral Bus). The bridging between AHB and APB is defined by ARM on the standard [16, Chapter 4.1].

On the used SELENE SoC configuration the AHB bus is 128 bits wide, analogous to the AXI (Advanced eXtensible Interface) interconnect width. It supports split transac-

tions (The bus is not kept waiting for the response of a slave indefinitely, it is freed and taken back by the slave when it has its response).

The default arbitration policy of the SELENE SoC is Round Robin. On the AHB bus every master has a HBUSREQ line to an arbiter, when they assert this line high, the master requests the bus. Then, the arbiter performs its arbitration and asserts high the HGRANT line, granting access to the bus to the slaves.

2.5.1. AHB Slaves

A slave responds to transfers initiated by masters in the system. The slave uses the HSEL signal from the decoder to control when to respond to a bus transfer.

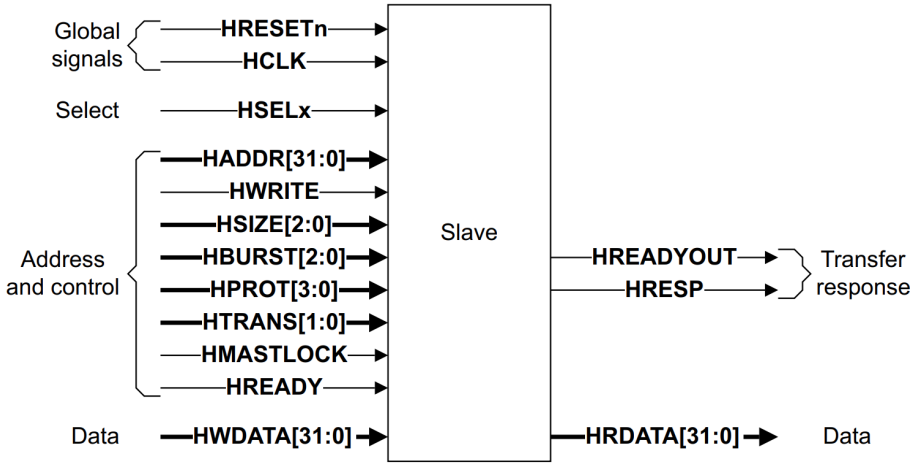

Figure 2.1: AHB slave interface

Source: [15, Figure 1.3]

The AHB slave interface is shown on Figure 2.1. Where the leftmost part are the inputs and the rightmost part are the outputs.

The AHB slave interface input signals are as follows:

- HReset: Resets the slave.
- Hclk: Clock to synchronize every bus participant.
- HSel: Signals whether the slave has been selected and should listen to the bus.
- HAddr: Represents the address that is being accessed on the bus.
- HWrite: When high represents a write transaction, when low represents a read transaction.
- HSize: Represents the number of bytes (+1) being transferred on each bus transaction.
- HBurst: Represents the burst mode of the transaction, where it can be an individual transaction, an incrementing burst of defined or undefined length or a wrapping burst. [15, Section 3.5]
- Hprot: Indicates how an access should be handled wihtin a system. This includes opcodes such as Data access, non-bufferable access, non-cacheable access...

- Htrans: Indicates the transfer type of the current transfer. This can be IDLE, BUSY, NONSEQUENTIAL and SEQUENTIAL.
- HmastLock: When high indicates that the current transfer is part of a locked sequence. This indicates that this transfer sequence is indivisible and must be processed before any other transfers. This is typically used to maintain the integrity of semaphores.
- Hready: When high indicates that the previous transfer is complete.
- HWData: Bus where the data is transferred. In the SELENE SoC case it is 128 bit wide, but on Figure 2.1 it is just 32 bits wide.

2.5.2. AHB Masters

An AHB bus can have one or multiple masters. When only a master is present, there is no need for an arbiter and the system only needs a Decoder and a Multiplexor for decoding the slave of an address and multiplex the read data bus and response signals from the slaves to the master.

When multiple masters are to be present, an arbiter and an interconnect is needed due to the bus non-tristate implementation. For the arbitration of multiple masters, a request line is connected to the arbiter, where the requesting masters are arbitered according to the AHB arbiter policy. The arbitration policies of this AHB arbiter will be the focus of this work.

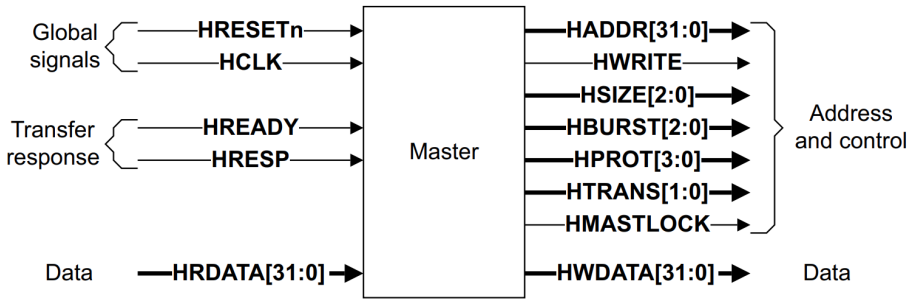

Figure 2.2: AHB master interface

Source: [15, Figure 1.2]

The AHB master interface input signals are as follows:

- HReadyout: Used to signal that a transfer has finished on the bus when high. When low it extends a transfer.
- Hresp: Provides the master with additional information on the status of a transfer, if low it indicates a correct transfer, when high it indicates an error.

2.5.3. AHB operation

Figure 2.3 shows the operation of a system with one master and three slaves. On this system the master starts a transfer by driving the address and control signals shown on Figure 2.2. Every transfer consists of several phases, for a system with several masters it has the following phases:

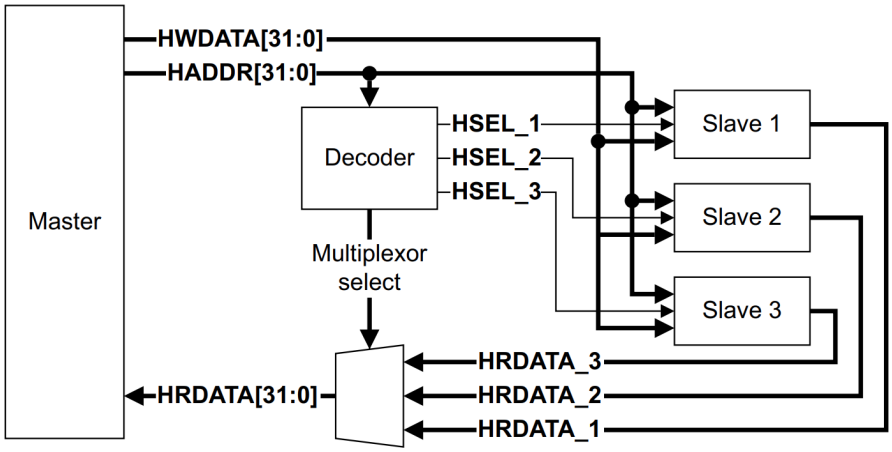

Figure 2.3: AHB block diagram

Source: [15, Figure 1.1]

- Arbitration phase: The master drives the request line until it receives a grant signal from the arbiter.
- Address phase: The master drives the address and control signals to perform an operation, the selected slave is decoded from the target address.
- Data phase: The master performs the data transfer.

This phases can run in parallel making the bus lines "pipelined" and thus, being the theoretical maximum throughput of the bus equal to one transaction each cycle with N masters. This throughput will be affected on a real system by the latency of the memory that is being accesed by means of this bus.

If no split transactions are used and the AHB slaves being accessed have a high access latency (Such as a DRAM) there will be a lot of idle cycles. AHB implements a functionality called split transactions, where one master can request the bus and be told by the arbiter that its response will come eventually (By issuing an SPLIT command).

When the master receives the split it understands that other masters will be granted the bus until its response is met. It needs to be said that this will only improve the utilization of the bus, not the throughput of the master, as the master cannot issue any more transactions until the response to the split is issued.

CHAPTER 3
Baseline System-on-chip

In this chapter, an overview of the baseline SoC will be given. The described on this chapter will serve as the base to introduce our modifications as will be explained on the next chapter.

Further explaining the inner working of the integrated BSC PMU that we used as a tool which functionality we leveraged to implement our modifications.

3.1 SELENE SoC

The SELENE SoC is the SoC used on the SELENE project. This SoC is widely parameterizable with number of cores, L2 cache size, associativity and memory access protocol (Currently we use AXI for memory connections, but AHB output is possible).

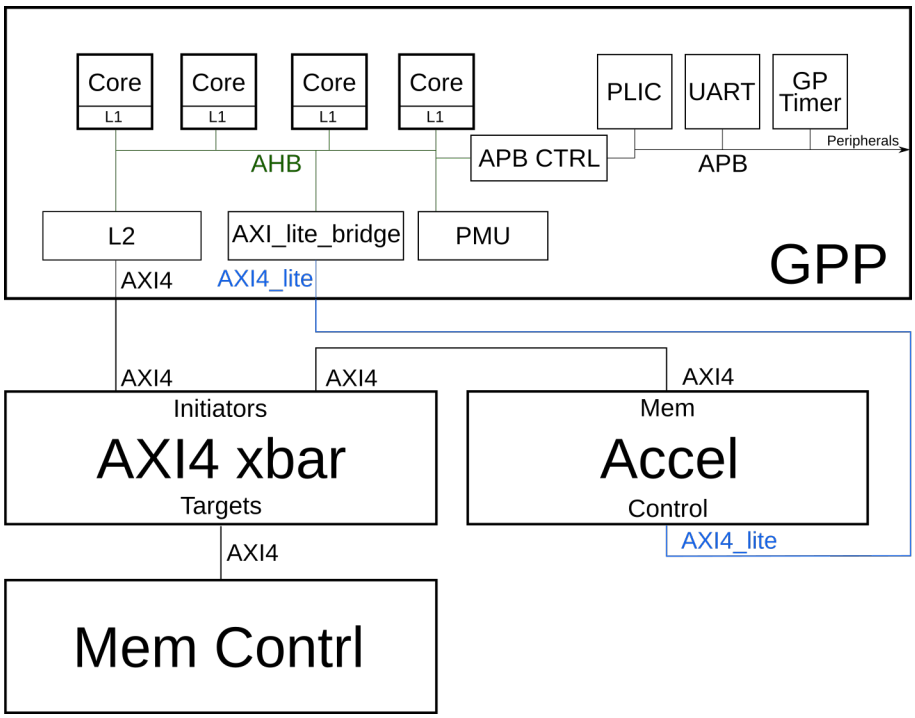

Figure 3.1: SELENE SoC

The SELENE SoC is highly configurable and makes use of the Noel-V Cobham Gaisler cores and core infrastructure. It makes use of all of Gaislers control libraries and plug&play interfaces to ease the interconnection of peripherals to the system.

The memory interface used on our configuration is the Xilinx MIG DDR4 controller IP for the VCU118 board. This controller is accessed by a 128 bit AXI interface. The memory controller's AXI interface is accessed through the AXI crossbar and subsystem as shown on Figure 3.1. The crossbar used is the one proposed on [17]. The default AXI crossbar and AHB arbitration policies are round robin, but further AHB arbitration policies will be studied on this work.

The SELENE SoC also integrates machine learning accelerators to greatly speed up neural network inference workloads. This results on a greater utilization of global resources and a more efficient coexistence of criticality workloads. This offers great advantages given a proper hardware support for those mixed criticality workloads. This hardware support at the AHB level is presented on this paper. Hardware support at NoC level is being developed at the writing of this paper and is defined as one of the SELENE H2020 project objectives defined at [18].

With the successful integration of machine learning accelerators into the SELENE SoC subsystem it will become a flexible system for real time and critical applications such as self-driving cars. With the appropriate SELENE SoC configuration and using the results of this work it becomes possible to have, on the same system, AI inference, critical task control and non-critical control all on the same chip. This greatly improves the system flexibility, size and power consumption characteristics.

For simulating the SELENE SoC we use a perfect AXI4 memory model instead of the MIG controller model (A realistic memory model of the memory controller used on the VCU118 FPGA). Some effort is being put currently to adapt a more realistic memory model such as the micron MT40A256M16GE-083E IT DDR4 simulation model presented on [19]. This usage of the perfect AXI4 memory model has the purpose of speeding up simulation for non memory critical debugging.

3.1.1. Noel-V processor

The cores are Gaisler NOEL-V cores on their GPP64 configuration [20], these are dual issue superscalar RISC-V cores that support the RISC-V IMAFD extensions. This extensions are the base Integer instructions (I), the Standard Extension Integer Multiply and Divide (M), Standard Extension Atomic Instructions (A), Standard Extension Single-precision Floating Point (F) and Standard Extension Double-precision Floating Point.

Inside the GPP, each core have private L1 data and instruction caches with LRU (Least recently used) and write-through policies. All L1 caches support AHB bus snooping, ensuring cache coherency between all private L1 caches in an individual GPP element. This cores are connected with a shared AHB bus, this AHB bus will be the subject of study on this work. The AHB bus is connected to the L2 cache and bridged over to the APB bus that is used to connect the peripherals. This organization can be seen on Figure 3.1.

The GPP element is essentially a NOEL-V subsystem. The basic contents of a GPP element are the Noel-V cores, with their respective caches. Currently the SELENE SoC uses a 6 core per GPP configuration. But on this work, a GPP configuration with 4 cores was used to simplify the system.

Each GPP element has an L2 cache. L2 caches support both write-through and write-back policies to reduce the traffic on the NoC. The SELENE SoC does not guarantee coherency between L2 caches in hardware. This means that, functionally, each GPP element is its own processor subsystem without any cache coherency nor low latency communication between GPP elements.

In addition to the NOEL-V cores, the GPP elements include debug support and system peripherals such as console UART, timers and performance monitoring.

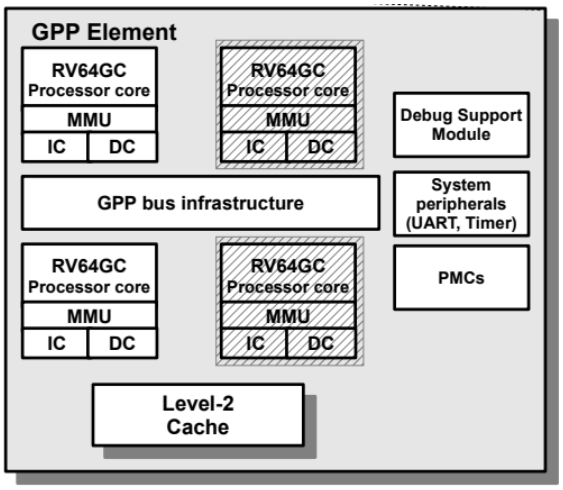

Figure 3.2: SELENE SoC GPP contents, internal schema. Source: Cobham Gaisler

An overview of the GPP system can be seen at Figure 3.2. Our modifications will be performed at the AHB bus, this AHB bus is located at the GPP bus infrastructure on Figure 3.2 and at AHB bus on Figure 3.1

Configuration	Setting
Number of GPP elements	1
Number of CPUs	4
Data cache size (kiB)	16
Instruction cache size (kiB)	16
Cache write policy	Write-through
Cache replacement policy	LRU
Floating-point unit	NanoFPU
GPP AHB data width (bits)	128
AXI data width (bits)	128

Table 3.1: SELENE NOEL-V GPP configuration

Finally the APB (Advanced Peripheral Bus) has the UART debug link and PLIC interrupt controller attached. Those two entities will be used by this work. The APB CTRL acts as a master on the AHB bus and has a high bus index, this high bus index is of special importance on the AHB static arbitration policy as higher indexes will get arbitrated first.

3.2 BSC PMU

"The SafePMU (Safe Performance Monitoring Unit) is an AHB slave capable of monitoring SoC events, enforce contention control and identifying profiling errors on runtime."[21]

The SafePMU attaches to a 128-bit wide AHB bus but only supports single accesses and burst 32-bit accesses. AHB lock accesses and protection control are not implemented. Fortunately the SELENE SoC does not have implemented yet lock accesses and protection control.

The PMU features that we used on this work are:

Index	Name	Type	Source	Description
0	'1'	Debug	Local	Constant HIGH signal, used for debug purposes or clock cycles
1	'0'	Debug	Local	Constant LOW signal, used for debug purposes
2	pmu_events(0).icnt(0)	Pulse	Core 0	Instruction count pipeline 0
3	pmu_events(0).icnt(1)	Pulse	Core 0	Instruction count pipeline 1
4	pmu_events(0).icmiss	Pulse	Core 0	Instruction cache miss
5	pmu_events(0).dcmiss	Pulse	Core 0	Data cache L1 miss
6	pmu_events(0).bpmiss	Pulse	Core 0	Branch Perdictor miss
7	ccs_contention(0).r_and_w	CCS	Core 0	Contention caused to core 0 due to core 1 AHB read or write accesses
8	ccs_contention(0).read	CCS	Core 0	Contention caused to core 0 due to core 1 AHB read accesses
9	ccs_contention(0).write	CCS	Core 0	Contention caused to core 0 due to core 1 AHB write accesses
10	ccs_latency(0).total	CCS	Core 0	-
11	ccs_latency(0).dcmiss	CCS	Core 0	-
12	ccs_latency(0).icmiss	CCS	Core 0	-
13	ccs_latency(0).write	CCS	Core 0	-
14	pmu_events(1).dcmiss	Pulse	Core 1	Data cache L1 miss
15	ccs_contention(1).r_and_w	CCS	Core 1	Contention caused to core 0 due to core 2 AHB read or write accesses
16	ccs_contention(1).read	CCS	Core 1	Contention caused to core 0 due to core 2 AHB read accesses
17	ccs_latency(1).total	CCS	Core 1	-
18	ccs_latency(1).dcmiss	CCS	Core 1	-
19	ccs_latency(1).write	CCS	Core 1	-
20	pmu_events(2).dcmiss	Pulse	Core 2	Data cache L1 miss
21	ccs_contention(2).r_and_w	CCS	Core 2	Contention caused to core 0 due to core 3 AHB read or write accesses
22	ccs_contention(2).read	CCS	Core 2	Contention caused to core 0 due to core 3 AHB read accesses
23	ccs_latency(2).total	CCS	Core 2	-
24	ccs_latency(2).dcmiss	CCS	Core 2	-
25	ccs_latency(2).write	CCS	Core 2	-
26	pmu_events(3).dcmiss	Pulse	Core 3	Data cache L1 miss
27	ccs_contention(1).write	CCS	Core 1	Contention caused to core 0 due to core 2 AHB write accesses
28	ccs_contention(2).write	CCS	Core 2	Contention caused to core 0 due to core 3 AHB write accesses
29	ccs_latency(3).total	CCS	Core 3	-
30	ccs_latency(3).dcmiss	CCS	Core 3	-
31	ccs_latency(3).write	CCS	Core 3	-

Table 3.2: SELENE default input events

Source: [21, Table 2.1]

- Self-test: Used to test the PMU functionality with known outputs for a given input.
- Counters: 32 bit synchronous hardware registers where input events are registered. On overflow they wrap around. Counters can be reset, read and written, allowing thus the setting of an initial value for this counters. They increment according to the event they are assigned to with the crossbar.
- Overflow: There is one, one bit overflow register per counter. When the counter overflows, the corresponding overflow register bit is set to one. Overflow bits are reset once the PMU is reset or a zero is written on its register. The overflow bits can be enabled or disabled by setting to one or zero the bits on the overflow enable register.
- Crossbar: Allows to route any input event to any counter. The input events are the ones shown on Table 3.2 and the counters are the ones previously mentioned. This crossbar is of special utility due to having less physical counter registers than logical events. On the default SELENE instance only 24 counters are instantiated to track 32 events. The physical register location is also important due to the MCCU configuration.
- MCCU (Maximum Contention Control Unit): The MCCU is the main enabler of this work. It is a unit that measures the contention for each core, rising a hardware

interrupt after a threshold of contention is exceeded. This threshold of contention is configured via one register per core, we will call those quota ceiling registers "quota limit registers".

The PMU is connected to the SELENE SoC as a memory mapped AHB slave, so, to access any of the PMUs contents, the PMU address needs to be known and contention can be created by accessing the PMU contents, but there is no need to traverse the L2 cache to access it.

The PMU is composed of several registers to configure and provide the desired functionality. Some registers were already mentioned with the Crossbar configuration registers, Overflow configuration and data registers, counter registers, MCCU weight and control registers.

3.2.1. MCCU

The MCCU is widely used for the policies that we developed with this work. This MCCU enables us to track the different counter progression and assign those counters different weights so they progress asymmetrically to the ceiling. The value of this counters times its weight is subtracted to the remaining quota after each cycle.

So, each cycle the MCCU will perform the following steps for each cores quota (the steps shown are for core zero):

- Multiply the event of its two counters times the weight of each counter:

$$Counter[0] * weight[0] + Counter[1] * weight[1]$$

- Take the remaining quota for this core and subtract the result of the previous step:

$$remainingQuota[0] - (Counter[0] * weight[0] + Counter[1] * weight[1])$$

- If the remaining quota is depleted set the core corresponding MCCU interrupt line to high until restart.

An example of operation with the MCCU is the following:

We want the core one to receive an interrupt when it has exceeded 30 cycles blocking the core 0 on the AHB bus. Thus, we assign to counter two and three (counters that are connected to the MCCU interrupt line that connects to core one) the input events eight and nine, that correspond to core one blocking read, and blocking write accesses from core zero (See Table 3.2. Then we assign a one to the weights of counters two and three and assign a 30 to the core one quota limit.

3.3 AHB arbiter default arbitration policy

The default AHB arbiter arbitration policy is round robin. In this section the implementation of the round robin policy on the SELENE SoC will be explained showing an example of several arbitration rounds. Bus arbitration policies are a good starting point to guaranteeing quality of service to critical workloads.

Bus Entity	UART	PLIC	Core 3	Core 2	Core 1	Core 0
Bus Index	5	4	3	2	1	0
AHB request vector	1	1	0	1	0	1

Table 3.3: Example of AHB bus arbitration, initial scenario

With round robin arbitration selected, the arbiter will check the AHB request vector and serve the next requesting master to the left, if there is no master at the left it will wrap around. To ease the comparison with Static arbitration lets choose the index 5 as the first request to be served. We will represent the last request served with the red color and the current serving request with green.

Bus Entity	UART	PLIC	Core 3	Core 2	Core 1	Core 0
Bus Index	5	4	3	2	1	0
AHB request vector	0	1	0	1	0	1

Table 3.4: Example of RR AHB bus arbitration, cycle 1

Once the index 5 request is served, we will move on to serve the request of core 0, as seen on Table 3.4. This is the Round Robin policy, this policy gives the priority to the next master from the last served request. In this case, core 0.

Bus Entity	UART	PLIC	Core 3	Core 2	Core 1	Core 0
Bus Index	5	4	3	2	1	0
AHB request vector	0	1	0	1	1	0

Table 3.5: Example of RR AHB bus arbitration, cycle 2

Once the request on core 0 was arbitrated on Table 3.4, the arbiter moved on to serve core 2 request, and while serving core 2 request, core 1 made one request, now there are two requests pending.

Bus Entity	UART	PLIC	Core 3	Core 2	Core 1	Core 0
Bus Index	5	4	3	2	1	0
AHB request vector	0	1	0	0	1	0

Table 3.6: Example of RR AHB bus arbitration, cycle 3

But, as our policy is round robin, this means that the next master at the left of the previously served must be served, so the PLIC request is served, while core 1 requests waits for its round of arbitration.

As seen in the previous figures, round robin is the fairest policy, making sure no master requesting the bus is starved. But, by being fair it makes the worst case of arbitration policy that you will be the last in line. For this example, the worst case would be to wait for 4 arbitration cycles until your turn is met, even though core 0 is supposed to be the highest priority core.

3.4 Xilinx VCU118

The FPGA we are using to prototype our design and as a target for the SELENE H2020 project is the Xilinx VCU118 Evaluation board from Xilinx. This FPGA has two 4GB DDR4 memory interfaces and is one of the larger FPGAs on the Xilinx Virtex Ultrascale+ group.

The VCU118 counts with 10/100/1000 Mbps Ethernet (SGMII), UART and JTAG over USB connectivity. This connectivity is leveraged by our SoC as the VCU118 is a target for the SELENE H2020 project and thus, has support for its features with the GRMON debugger.

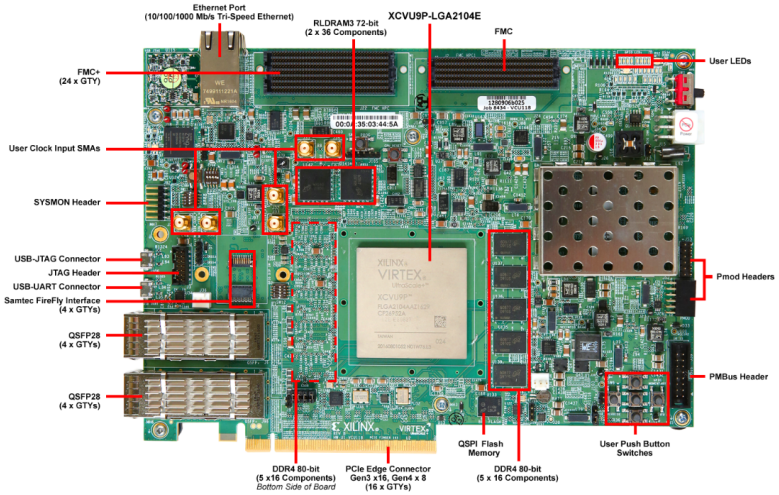

Figure 3.3: Xilinx VCU118 board

This FPGA counts with 2,586,000 System logic cells, 6840 DSP slices, 345.9 Mb of memory, 120 GTY 32.75 Gb/s Transceivers and 832 IO pins. The VCU118 board can host the latest SELENE SoC with its 6-core configuration and one small accelerator in just 12% of its area, so it was perfect for this application, as it is expandable for multiple GPP configurations and larger accelerators.

CHAPTER 4

Improving performance Guarantees in the SELENE SoC on-chip Bus

In this chapter the modifications done to the SELENE H2020 SoC are presented. The presented modifications have been implemented on the SELENE SoC and the evaluation of the proposed modifications will be presented in the next chapter.

The purpose of these modifications is to improve the performance of the critical and non-critical core workloads performance on the SELENE system while having a bounded execution time for the hard real time tasks.

These modifications are performed on two hardware modules of the SELENE SoC: the Cobham Gaisler AHB arbiter and on the PMU. Actually, one of the goals of this work consists on binding the PMU and AHB arbiter functionalities. The result of these modifications are a new set of shared resources arbitration policies that can be used to adapt the system to the need of a diverse range of real-time applications.

4.1 Improving on-chip arbitration policies

The objective of this work is to bound the execution time of critical workloads on the SELENE SoC. The execution time of core 0 (the critical core) workloads should be not significantly affected by the presence of hardware resources that introduce jitter. As the SELENE SoC is a multi-core system, a critical workload should share both subsystems.

On the cache subsystem, static partitioning can be implemented by giving each core its private L2 cache portion so no interference is possible, and, on the AHB bus, some modifications to the default round robin can be implemented to give priority to the priority core.

With the round-robin mode, the bus arbitration priority is rotated after each AHB transfer. If no AHB master is requesting the bus, the last owner is granted (bus parking) [22, AHBCTRL]. We will not contemplate bus parking on this examples for the sake of simplicity.

The round robin arbitration policy is the default arbitration policy for the Noel-v cores, but there is a static policy that can serve us as a way of prioritizing the critical core over the non-critical ones. This prioritization would reduce the critical core WCET and is the simplest way to ensure the maximum efficiency of the main core. The aforementioned static policy is explained on the following section.

4.1.1. Modified static policy

With the default AHB controller static arbitration mode, the master with the highest bus index has the highest priority. The problem is that the core with the lowest bus index is the core 0, that is, the core that we want to have the highest bus priority.

We want the core 0 to have the highest priority as it is the one that has the most contention metrics available on the PMU (as seen on Table 3.2). To achieve this objective we modified the default bus priority as described in the following paragraphs.

With our modified static bus priority the highest bus index still has the highest priority, but, now between the lower 4 bus indexes, the lowest bus index has the highest priority. This is done to give core 0 the highest priority between all the cores, but still give the interrupt controller and UART debug links priority over core 0 on the bus.

This new scheme of giving priority to AHB indexes greater than 3 to the bus over the index 0 (core 0) is done to preserve the UART debug link and interrupts working. We discovered that by merely reversing the default static priorities (lower index first) the UART link becomes starved of bus cycles when all four cores are running at full speed, making it impossible for us to debug the running programs and also starving the interruption controller from raising interruptions.

With that in mind, we implemented the new static policy. This new static policy consists on keeping the high index masters having the most priority, then, between the cores, the core with the lowest index number has the highest priority. From now on in this work, when we refer to static priority, we are referring to our modified static priority, not the default static priority previously described.

4.1.2. Example of arbitration with Static policy

With static arbitration the order of arbitrating between requesting bus masters is simpler than with Round Robin. With round robin, the previous arbitration results determined the next ones, thus, even if is fairer than static policy, pathological cases still appeared.

With our static arbitration we give priority to the highest bus index master except between masters [0-3], where we give priority to the lowest bus index master. Thus, not starving IO nor interruptions.

Bus Entity	UART	PLIC	Core 3	Core 2	Core 1	Core 0
Bus Index	5	4	3	2	1	0
AHB request vector	0	1	0	1	0	1

Table 4.1: Example of RR AHB bus arbitration, cycle 1

On the first round of arbitration consider the example shown on Table 4.1, the bus is given to the PLIC (Bus index 4) as it is the highest index master that is requesting it.

Bus Entity	UART	PLIC	Core 3	Core 2	Core 1	Core 0
Bus Index	5	4	3	2	1	0
AHB request vector	0	0	0	1	1	1

Table 4.2: Example of RR AHB bus arbitration, cycle 2

In contrast to Table 4.1, on the arbitration round presented on Table 4.2 the bus is given to the lower index master, that is, to the core 0, according to our newly implemented static arbitration.

Bus Entity	UART	PLIC	Core 3	Core 2	Core 1	Core 0
Bus Index	5	4	3	2	1	0
AHB request vector	0	0	0	1	1	0

Table 4.3: Example of RR AHB bus arbitration, cycle 3

Finally, on the last round of arbitration, the bus is given to the core 1 request due to it being the lower index and not having any requests on indexes [4-5].

This static arbitration approach allocates the resources unfairly by design, giving priority to the master with more priority. This means that, if the PLIC (Interrupt controller) was to keep requesting the bus each cycle, every core would potentially be starved forever.

4.2 The hardware Quota mechanism

The BSC PMU MCCU subsystem implements a quota mechanism, this quota mechanism raises an interrupt line when the quota is met. This interrupt line should be handled by the Interrupt controller and then, the contending processor core. The quota mechanism was explained on subsection 3.2.1.

This software approach depends on the operating system handling the interruption on a timely manner. Currently, the quota interrupt is routed to the contending core. With it, once the interrupt is raised, it arrives to the interrupt controller, that puts it in a priority queue with other pending interrupts. From there, the interrupt line to the corresponding core is raised to high and the core will perform the interrupt handling process.

With this scheme, the non-priority core that is causing contention needs to handle its own exception, meaning that it should stop itself from using the bus until further notice. This forces a case where, if the interrupt was not handled correctly or timely, the contending core would still cause contention and violate WCET. To avoid this problem, the interrupt handling software should be handled as critical, but this would force us to execute critical software on non-critical cores, over-complicating the system.

Another possible software solution is to route the MCCU exceeded quota interrupts to the critical core. With this approach, all critical workloads will be isolated to the main core, but another problem arises. Each time a contending core needs to be disabled the main core should run an interrupt handling process that should be scheduled and taken into account for the WCET. When executing the interrupt handling process, a change of context should be performed by the critical core, polluting the cache and wasting cycles on stopping the secondary cores.

With our hardware implementation of the quota mechanism, we mask the AHB request lines going into the AHB arbiter so no possibility of contention exists. We leverage the existing PMU infrastructure to achieve this objective by rerouting the previous quota interrupts into the AHB arbiter.

The quota mechanism lets us mask the memory requests of each contending core, effectively stopping them without explicitly doing so. This hardware quota mechanism is almost instantaneous, as it takes on the worst case one arbitration round to stop the contending core.

In contrast with the software approach, this hardware approach does not generate any contention, does not require the main core workload to be halted and does not force non-priority cores to perform a critical task, greatly simplifying WCET and pWCET calculations for the main core and simplifying the system itself.

4.2.1. Quota mechanism example

An example of this quota mechanism is presented on the following diagram:

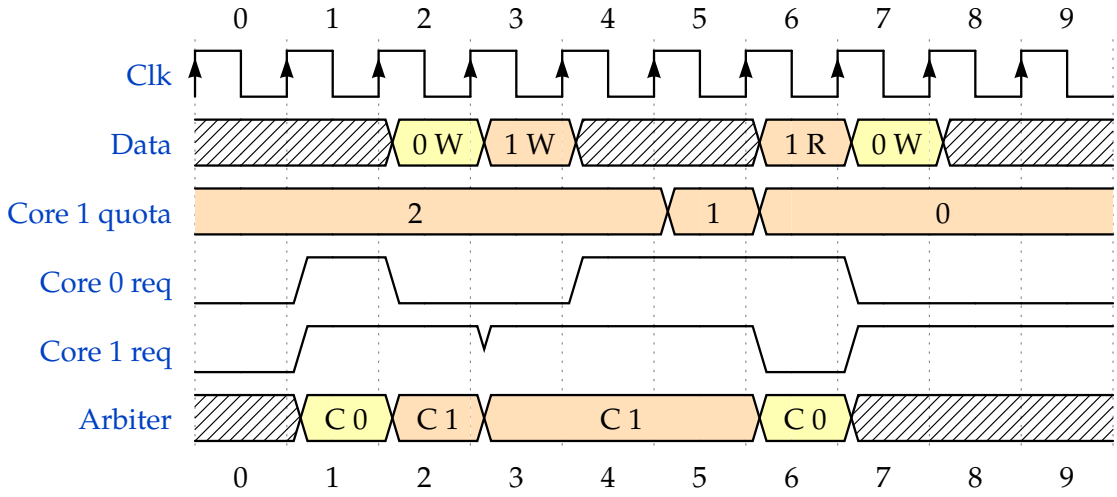

Figure 4.1: Example of the Quota mechanism

To demonstrate the quota mechanism working, we have the scenario described on Figure 4.1. It initially has the core 1 quota set to 2. It has the weight registers described on subsection 3.2.1 set to one and the counter registers set to 7, or "Contention caused to core 0 due to core 1 AHB read or write access" Table 3.2.

With this setup, when the system is initialized, the core 1 can block the AHB requests from core 0 a total of two cycles. Once the core 1 quota reaches zero, its requests will be masked to the AHB arbiter and it will not be able to perform any more requests until the MCCU is reset or the quota is updated.

This masking lets us preserve the independence of the already validated AHB arbiter but add the described functionality to it. In this way, we simplify the arbiter mechanism so it does not take into account special edge cases.

On cycles 4 and 5 the core 1 request is being fulfilled while core 0 is requesting the bus, so the core 1 quota is decreased. On cycle 6 the bus is given to the backed up request of core 0 and on cycles 7, 8 and 9 the bus is not given to core 1 even though it is the only one requesting it due to our quota mechanism having reached zero.

4.2.2. Summary

With the hardware quota mechanism the maximum contention that a task in a priority core can incur as a result of sharing the AHB bus can be bounded to a known, software defined value.

This bounding of the contention (or increase of execution time) of the primary core workload does not necessarily come at the cost of decreasing the performance of secondary cores that share the bus with this critical core. In fact, on a time period where secondary core accesses to the bus do not cause any contention to the primary core, the

quota system will not block petitions and secondary core progress will be the same as if their workloads where running on a system with no quota.

To summarize, the quota mechanism guarantees a maximum allowed interference for the critical core while doing a best effort to guarantee progress on the secondary cores.

4.3 Improving the hardware quota mechanism

With the previously described quota mechanism, there needs to be an explicit API call to reset the quota counter, and thus, restart the stopped cores. This increases our execution time and, in a time-composable system, needs to be scheduled accordingly. This call can be either at the end of the critical task before beginning the next one or each predefined time period.

For less stringent real-time systems such as some automobile systems a self-resetting quota mechanism proves useful as it occurs predictably and its effect on other tasks can be bounded over a large period of time. This is only feasible on applications with some flexibility in the timing requirements. For some highly critical control tasks time predictability of each task is crucial, and, even-though over a large period of time the interference is bounded, it is very hard to prove that its bounded on each scheduling period.

4.3.1. The quota refill mechanism

For soft real-time applications we implemented a quota refill mechanism that refills the core quota each predefined time period. Therefore, over a large period of time the maximum contention is bounded to our software-defined value.

This bounds the maximum possible interference to the main core to a percentage of its total execution time while not imposing any additional software overheads each scheduling period and, on the best case, letting all cores run at their maximum alone theoretical speed.

Over the next subsection we will present an example of the refill mechanism as well as show how it is integrated into the PMU subsystem, how it is interfaced with and showcase the improvements over the previously described quota.

Example of quota refill mechanism

To illustrate the quota refill mechanism on the AHB bus a small example with 20 cycles of transactions is created on Figure 4.2 and Figure 4.3.

On figures Figure 4.2 and Figure 4.3 the following signals are present:

- Clk: Represents the clock signal.
- Data: Represents the AHB data bus. In reality this is a much more complex system, but we are concentrating all data signals on data and all arbiter signals on "arbiter".
- Quota refill period: Represents the register pictured on Figure 4.6. The number represented here is configured by software and represents the number of elapsed cycles between when the quota of the core is exhausted to where is reset.
- Core 1 refill counter: Internal counter that starts once the cores quota is exhausted and is reset once it is restored. This counter is implemented in hardware and is not

visible to the end user. On a multi-core system there is one of these counters per core.

- Core 1 quota: Represents the remaining quota for core 1. This value is initially the one set on PMU register "0x07c" and decreases automatically with the formulae described on subsection 3.2.1.
- Core 0 req: Request line going from core 0 to the AHB arbiter, when the signal is high core 0 is requesting access to the bus.
- Core 1 req: Request line going from core 1 to the AHB arbiter, when the signal is high core 1 is requesting access to the bus.
- Arbiter: This signal represents the arbiter decision on the AHB bus. This signal is a simplification of the real AHB signals described on section 2.5, although this is a simplification it still represents the two phase arbitration system correctly.

The example system described here has the following values set:

- QuotaRefill: With a value 5. This means that the quota of a core is replenished 5 cycles after it is met.
- Core 1 quota: The value for the core 1 quota is 2. This value is written into the core 1 quota register after the quota refill period is met and initially, when the PMU is restarted.
- Event weights: For core 1, we configured the PMU so that each cycle that core 1 blocked core 0 requests, one point was deducted from its quota. This means setting up the counter[2] (mapped to core 1) to track the event 7 (Contention caused to core 0 due to core 1 AHB read or write access)Table 3.2. Then, we set the weight[2] register to 1 and the weight[3] register to zero as we don't want to track the value of any other counter.

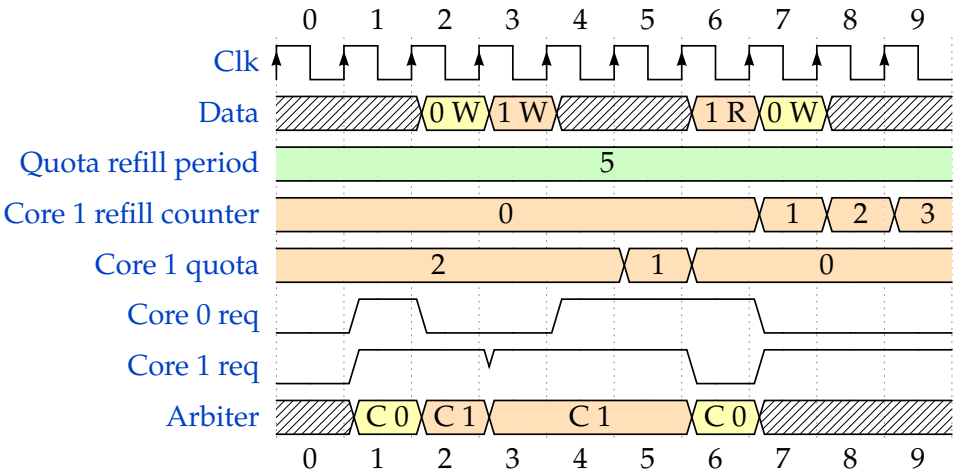

Figure 4.2: AHB quota refill example, part 1

With these values, we can start to analyze the example shown on Figure 4.2. As previously mentioned, the initial quota is 2, so on cycle 0 the core 1 quota is 2 as the core 1 has not caused any contention yet.

Then on cycle 1, both core 0 and core 1 request the bus and the request is granted in a combinational manner to core 0 as is the priority one. On cycle 2, the write request of

core 0 is served on the data line and the request of core 1 is granted as it is the only one requesting the bus.

On cycle 4, while the read request of core 1 is being served, core 0 requests the bus, but is blocked, as the request of core 1 was already granted and is being served. The core 0 request remains blocked until cycle 6, where it is granted, as the previous request from core 1 was met (as seen on the data signal).

Cycle 6 also illustrates what happens when the quota of a core is met. As core 1 blocked two cycles core 0, its quota got depleted, and thus, its requests are masked. Now, as the core 0 quota is depleted, the quota refill mechanism counter starts counting up until the value of "Core 1 refill counter" is the same as the value of "Quota refill period". Without the quota refill mechanism the core 1 requests would have remained masked until a software reset was issued by core 0.

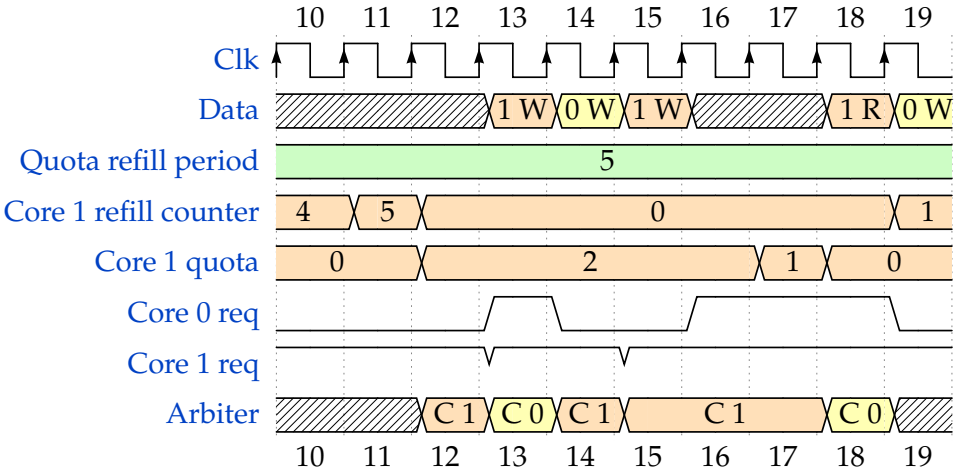

Figure 4.3: AHB quota refill example, part 2

Then, Figure 4.3 represents the continuation to the timing diagram presented on Figure 4.2. On cycle 11, the core 1 quota reaches the value on quota refill period, and thus, on cycle 12, the core 1 req is unmasked and the arbiter can see it on its arbitration round. Also, the Core 1 refill counter and Core 1 quota counters are reset to their starting values.

On cycle 13, the core 0 is selected on the arbitration round as static priority arbitration is being used. On cycle 14 and 15, core 1 is selected on its arbitration round. The core 1 cycle 15 request is a read request, so it takes 3 cycles to be served, thus, blocking core 0 for two cycles and repeating the previously described quota and quota refill mechanisms.

Quota refill PMU integration

To configure our created quota refill mechanism, we added some configuration registers to the preexisting PMU and MCCU configuration registers.

We first added a configuration register that can be seen on Figure 4.5 from the original Figure 4.4 register. This prompted us to adapt the PMU API to use 32 bit memory accesses instead of the previous 8-bit accesses, but allows us to further expand the unused configuration register to add more features in the future.

This adaptation to 32 bit memory accesses does not introduce any overhead as the AHB bus is 128 bits wide and we are performing 32 bit accesses. This means that the setup and transfer times of 32 bit wide accesses is the same as 128 bit accesses in bus time.

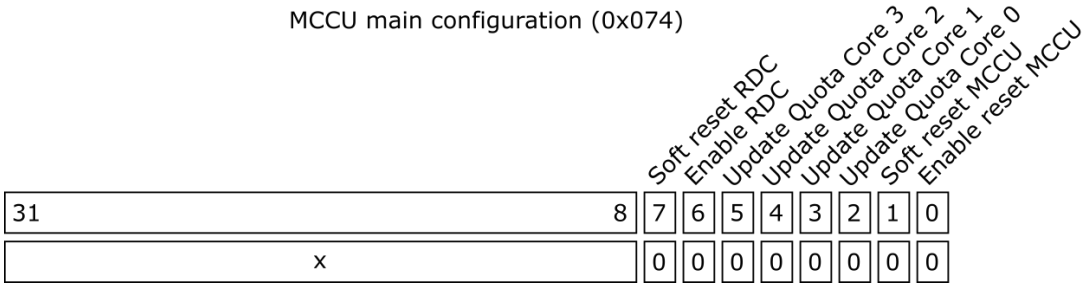

Figure 4.4: MCCU main configuration before quota refill

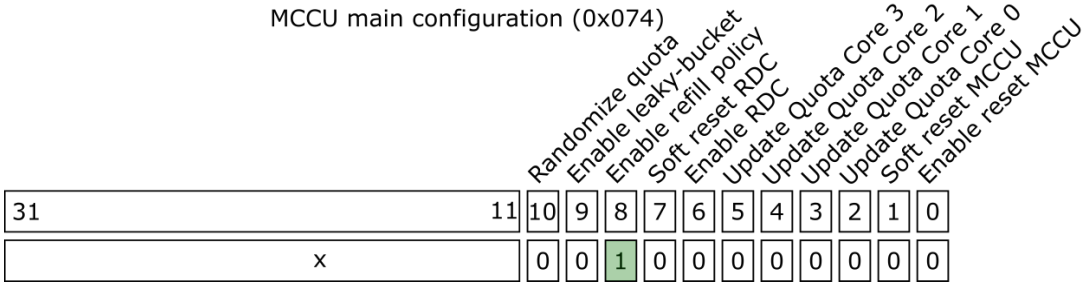

Figure 4.5: MCCU main configuration after quota refill

We also added an additional register to control the quota refill reset period as seen on Figure 4.6. The value of this register controls the number of elapsed cycles from where one core reaches its quota and is disabled to where it is granted its whole quota again.

With this quota refill mechanism, we can bound the contention caused from one core in a time period, and, by setting the quota refill counter from the moment each core finishes its quota, we can sort of randomize the quota reset time to avoid some pathological cases.

This quota refill approach can be fine tuned by software by balancing the ratio between the quota value and the refill value. This ratio determines the memory pressure that contending cores will generate.

Summary

The quota refill mechanism lets us tune the quota mechanism in a hardware manner but still being software configurable. It lets us restart the quota mechanism without wasting priority core cycles and lets us bound the worst case execution time of core 0 while still letting the other cores progress at a high rate.

The cooldown period of the quota is used to rate-limit the other cores requests, thus, bounding WCET. For example, in the case that our system had two cores (One priority and one not priority), the worst contention caused by core 1 would be:

$$\text{Core 1 max contention} \approx \frac{Quota}{Quota + QuotaResetPeriod}$$

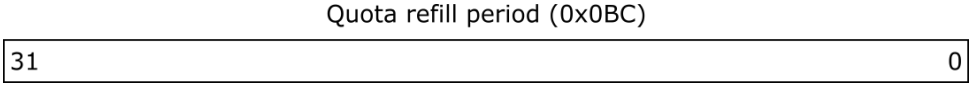

Figure 4.6: Quota refill register added to MCCU

This represents the maximum contention, and, for a system without mixed criticality this is the contention that should be scheduled. But in a dual-core system where mixed criticality is present, the core 0 is the critical core and core 1 is not the critical core. Using this mechanism, we can assure that the time that will take for the critical application to run with the core 1 workload (assuming perfect DRAM memory) is, at the worst:

$$Core0MaxExecutionTime = ExecutionTimeAlone * (1 + Core0MaxContention)$$

On a statically arbitrated system without mixed criticality, for a 10% contention, the core 1 workload would run a 10% of the time and multicore will be forbidden. With our method and on a system where the bottleneck is the AHB bus, we can ensure that the critical load worst case execution time (for a 10% max contention) is its execution time alone * 1.10. While we can assure that the execution time of the non-critical load is, at its worst, a 10% of its execution time alone. But, at its best, its execution time alone.

This assumptions are made only taking into account the bus arbitration times, for a non-ideal system where non-contiguous DRAM access heavily penalizes throughput the results would differ.

4.3.2. Randomized quota refill period

The previous implementation of the quota refill mechanism created pseudo-random refill periods that depend on the traffic generated. But those periods might create pathological cases where there is a systematic blocking of the main core or a case where WCET is violated systematically.

These pathological cases greatly diminish the effectiveness of the policies and might over punish one specific core due to its traffic pattern. This generates the "synchrony-effect" observed by [4, p. 4] on our system. With this effect we are susceptible to some patterns that heavily penalise one individual core with the RR and static policy. This patterns end up repeating due to traffic slowly synchronising itself to a given order.

To break with this synchrony-effect, we implemented the randomized quota refill. This mechanism works the same as the traditional quota refill, but randomizes the core initial quota each time the refill mechanism is triggered. This refill of the quota results on a new quota with a random number between (quota refill * 2, 0). This preserves the quota refill period fixed by software, so there is no perceptible change.

The randomization of the quota is performed with a 32 bit LFSR (linear feedback shift register) with an initial seed of the value of the quota of core 0. Then, the formulae used to obtain the FSR of cycle x+1 with the value of cycle x is: $LFSR(x+1)[0] = LFSR(x)[31] xnor LFSR(x)[21] xnor LFSR(x)[1] xnor LFSR(x)[0]$

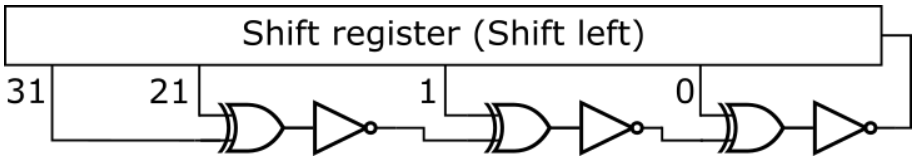

Figure 4.7: Our LFSR implementation

In Figure 4.7 we can see a mock-up of the LFSR we used on our design. This is a simple shift register that is fed back from a combination of its previous values creating pseudo-random numbers. The initial value of this shift register is the quota of core 0.

As stated, once the quota is met we latch the value of the LSFR and a mask for the bits used on the quota. We obtain the quota mask by using a cascade of or gates as shown

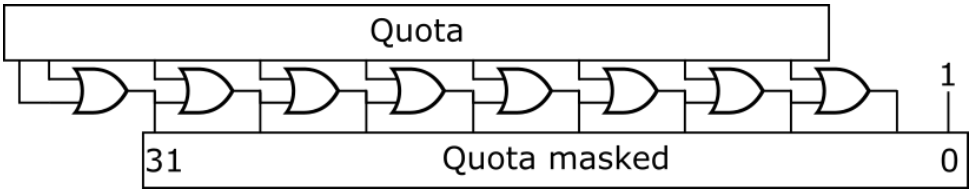

Figure 4.8: Quota mask for the LFSR register

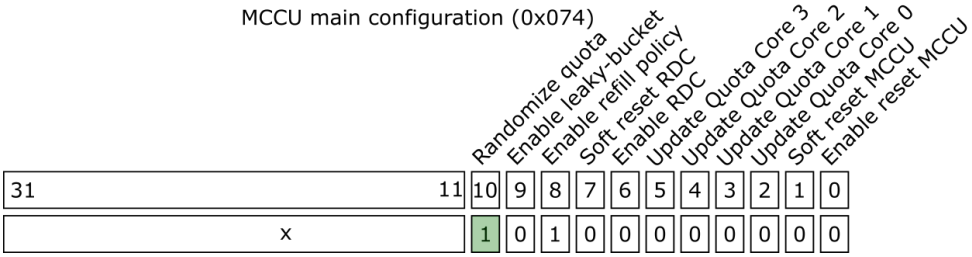

Figure 4.9: MCCU main configuration after randomized refill implementation

in Figure 4.8. This makes that the leftmost 1 trickles down to every bit under it. Thus, having the effect of rounding the result to its higher power of two.

As seen on Figure 4.8 the quota is shifted left by one position, this is done to multiply by two its value and make the average of the quota masked and LFSR the value of the quota. This averaging happens as the result of "LFSR" and "Quota masked" is a random number between (0, Quota*2).

Quota randomization PMU integration

In terms of implementation into the PMU, the quota randomization mechanism is built as an extension of the refill mechanism implemented on subsection 4.3.1. This randomization mechanism affects the refill period configured on the refill register Figure 4.6.

When the randomize quota register is set to 1 in the Figure 4.9, the refill period becomes a random number on the [QuotaRefillPeriod*2:0) range. This number is obtained using the mechanisms described on subsection 4.3.2 and varies each refill period of each core. This randomizing reduces the possible pathological cases and avoids bus synchronization if *QuotaRefillPeriod* and *Quota* are set correctly.

When the randomize quota register is set to 0 in the Figure 4.5, the refill period becomes *QuotaRefillPeriod* and is not randomized. This might be useful when a previous analysis of the application is performed and tight control over the refill periods is needed.

Summary

The randomized quota refill mechanism is a direct improvement over the previously described refill mechanism. It provides the same functionality and the same timing characteristics than the original refill mechanism while greatly improving its pathological scenarios, and thus, increasing its fairness on round robin and the prioritization on the static policy.

This makes the refill mechanism a useful rate-limiting mechanism for soft real-time systems, preserving the best possible execution time of the non-priority tasks while bounding the maximum possible contention that they can cause to the priority task.

4.3.3. The leaky-bucket mechanism

The refill mechanism described before works great with soft real-time systems where missing deadlines is not as critical as in hard real time systems. On hard real time systems where there are precise scheduling periods, a more exact quota mechanism must be put in place.

Lets put the example of a scheduling period of 150 cycles where the worst case execution time of the priority task is 100 cycles. This means that, over the aforementioned scheduling period, there are 50 contention cycles that we can give out to the non priority cores. Let's say that we decide that core 1 is the most priority core of the non-priority cores and we give it 30 contention cycles. Thus, giving cores 2 and 3 10 cycles each.

Core 0	100 Cycles WCET alone
Core 1	30 Cycles allowed contention
Core 2	10 Cycles allowed contention
Core 3	10 Cycles allowed contention
Total	150 Cycles

With this setup, we can ensure that the priority task will complete always on its worst case execution time and fill the scheduling period with other workloads. The usage of a quota also makes possible for the core 1, 2 and 3 workloads to progress more than their allowed contention cycles if they are not blocking the core 0 workload. In fact, this progress is almost guaranteed as the probability of a hit on the core 1, 2 and 3 private caches is high.

The premise of the leaky bucket mechanism is to leverage the existence of the private L1 caches and prevent starvation on the non-priority cores by slightly violating the total contention that they are allowed. This is obviously on violation of the strict WCET calculus, but can be on line with probabilistic WCET.

Let's put an example of a critical workload that should be scheduled for its worst case execution time, but only will reach its worst case 0.01% of the times. On its WCET it will take 100 cycles to complete, but on its typical time it will only take 30 cycles to complete. This critical workload will always check the value of some given sensors (30 cycles), and, if an anomaly is detected, take action (100 cycles).

Leaky bucket example, best case scenario

Finally lets also suppose that the core 1 workload pathologically blocks the critical workload 30 cycles on its first access to the bus, thus, reaching its quota. Lets also suppose that, over this scheduling period, core 2 and 3 never block core 0 workload due to them having a very high L1 cache hit ratio over this period of time.

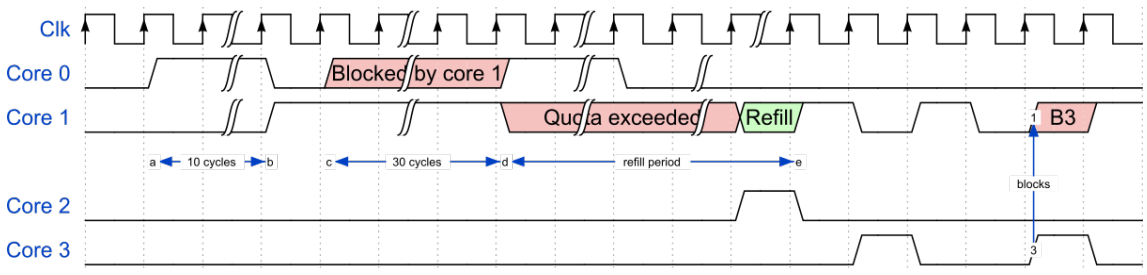

Figure 4.10: Leaky Bucket example, best case scenario

With these assumptions we can make the timing diagram shown on Figure 4.10. In this timing diagram, Core 1 exhausts its quota between points *c* and *d* with its quota being refilled on point *e*. This quota was originally 30 cycles, but is only refilled by one contention cycle, but, as core 0 workload already finished it will not be depleted until the end of the studied period.

On B3, core 3 access blocks core 1 from accessing the bus, but its quota is not decreased as it is not the priority core. This example was the best case, where core 0 only had to check its sensors and its execution time was much better than with the WCET.

In this example there is not much difference between the refill mechanism explained on subsection 4.3.1 and the leaky bucket. This is due to the leaky bucket mechanism being an update to the refill mechanism to cater to hard real time systems.

The aforementioned similarity is due to the leaky bucket mechanism being a partial refill with a much lower value to let the non-priority core progress. This lower value is configured at *QuotaLeakyRefillValue* PMU register located on *0x0C0* on the PMU memory space.

Leaky bucket example, worst case scenario

In this example, we will show the worst case scenario of the leaky bucket mechanism. In this worst case scenario we exceed the scheduling period without the critical workload finishing. This is due to our over-provisioning of resources and can be seen on Figure 4.11.

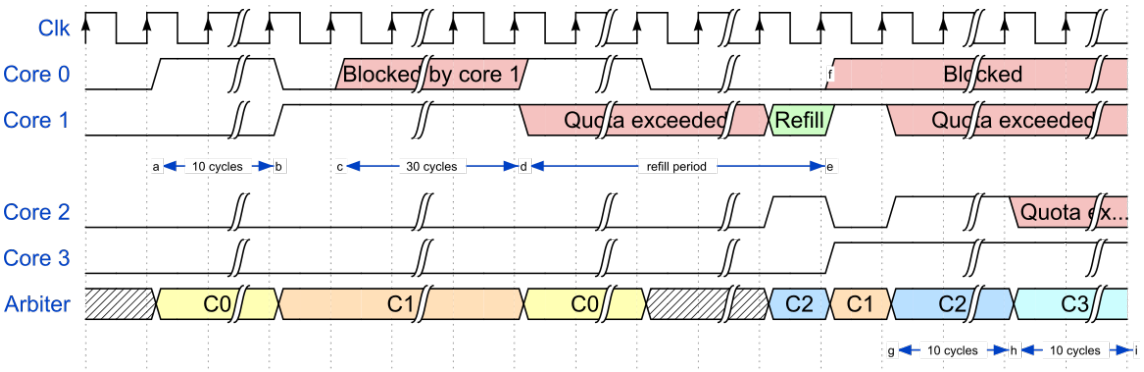

Figure 4.11: Leaky Bucket example, worst case scenario

For this leaky bucket worst case to occur on this example, the following conditions should be met:

- The scheduled critical workload should take all its worst case paths and reach its calculated WCET.
- Every core should reach its maximum scheduled contention.
- The leaky bucket mechanism should refill one contending core and this core should generate contention to the main core.

All this cases can be seen on Figure 4.11. Between points *c* and *d* core 1 exhausts its quota. Then, this quota is filled by one more cycle (Leaky bucket refill) on point e. This refilled quota instantly is used by core 1, then getting blocked again. After this quota is used up, cores 2 and 3 use their 3 cycle quota in the *g* to *i* period. When this events occur, the core 0 workload will exceed its WCET.

For some use cases, this excess of WCET is not critical if it does not present itself in a pathological manner. But, unfortunately without any randomization, pathological cases might occur.

Leaky bucket PMU integration

To implement the leaky bucket mechanism we modified the previously described refill mechanism by making the refill value configurable via the leaky bucket reset period register (Figure 4.13).

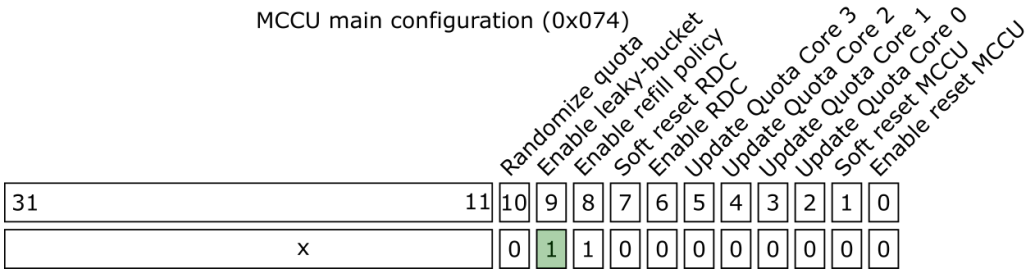

Figure 4.12: MCCU config register with Leaky bucket mechanism activated

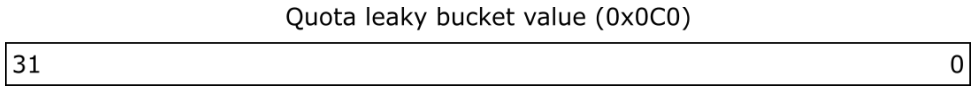

Figure 4.13: Leaky bucket register added to MCCU

This implementation of the leaky bucket mechanism as an extension of the refill policy means that, in order for the leaky bucket to work both the refill policy and the leaky bucket bits should be enabled in the Figure 4.12.

Once both bits are enabled, the refill register Figure 4.6 (0x0BC) should be used to control the frequency of the refill once the quota is exhausted. The *leaky bucket reset period* register (0x0C0) shown in Figure 4.13 should be used to control the amount of quota that is refilled each refill period.

During the development of this work on the SELENE H2020 project, an API was deployed to easily interface all the presented features on a user friendly manner. This API internally performs the aforementioned operations.

Randomized leaky bucket

To avoid pathological cases over several scheduling periods, we can apply the same randomization method that we previously applied to the refill mechanism to the leaky bucket mechanism. This is as simple as setting the Randomize quota bit to '1' on the MCCU main configuration (0x074) as well as the Enable leaky-bucket and refill policy bits to '1'.

Once this configuration is made, the leaky bucket refill period will be randomized on a scale of [QuotaResetPeriod:1]. With the randomized leaky bucket approach we can greatly mitigate the synchrony effect described on [4, Section 4], thus, breaking up memory patterns that might turn into pathological cases for the priority core.

Summary

The randomized leaky bucket mechanism is a direct improvement over the refill mechanism, letting us fine-tune the quota parameters to cater our application needs. The final system with leaky bucket and randomized periods should be able to not present pathological cases, not starve secondary core workloads and still present us with a mechanism to bound the maximum contention to our primary workload.

CHAPTER 5
Performance evaluation

In this chapter, we first introduce the hardware, the software and the tools that we used to obtain the results of the previously described mechanisms. A description of the hardware platform and how we interface with it will be provided. Next, the microbenchmarks used for the performance characterization will be laid out, and finally the results of our different arbitration and quota mechanisms will be presented.

5.1 Hardware platform

In this section, we describe the FPGA configuration that we used to obtain our results and the program that we used to interface with our FPGA board and, ultimately, with our design. The aim of this section is to allow the reader reproducing the obtained results on their board, as it is a commercial board and the proposed design will be open source by 2022.

5.1.1. GRMON

As stated in its documentation, "GRMON is a general debug monitor for the LEON (SPARC V7/V8) processor, NOEL-V (RISC-V) and for SOC designs based on the GRLIB IP library"[23]. Note that we use the GRLIB IP library to build the SELENE SoC. The rest of the components added to the SoC are also compatible with the GRLIB integration scheme and thus, the resulting SoC can be interfaced with GRMON.

We use the GRMON platform to debug our design. Our design is based on the NOEL-V cores and, thus, has the necessary debug interfaces to use with GRMON. GRMON can be used with an UART and Ethernet as debug link interfaces.

With the Ethernet debug interface, the process of uploading large binaries is much faster. For example, for uploading a Linux image binary, with Ethernet it is done in less than a minute while with UART it takes around 45 minutes. This is due to UART transfer speeds being around 100 kb/s while Ethernet's are around 35 Mb/s [23, Table 5.1]. In this work, we use the Ethernet interface extensively as it generates less contention on the studied AHB bus than the UART debug interface.

GRMON offers countless debugging options, but the ones that we have exploited are the memory proving tools. With this memory proving tools you can launch accesses to memory from the debug interface and get the value.

Memory proving is specially useful with memory mapped IO, such as the BSC PMU that was extensively used in this work. With the command "mem address bytes", a memory request is launched from the debug interface and the result can be printed on the

screen. We extensively use this command to generate scripts and launch our benchmark suite in an automated way.

We also used extensively the reg commands to read some of the RISC-V csr registers such as the total cycle counter to estimate the execution time of our benchmarks.

GRMON also offers several debugging and control parameters that are useful when configuring our scripts such as the ability to write on specific memory addresses and enable or disable several SoC features. The most useful features that we extensively used for this work where the ability to flush, invalidate, and enable-disable L2, L1D and L1I caches. This, with the ability to reset the system to prepare it for a new benchmark were the foundation of our testing on the real hardware.

GRMON also offers some debugging features that were used in this work for debugging purposes, such as setting breakpoints and debugging step by step with the usual debugging features of a modern hardware debugger.

5.1.2. Board configuration

The FPGA that we used for prototyping and as a target for the SELENE H2020 project is the Xilinx VCU118. This is one of the larger Xilinx FPGAs and is used to prototype the SELENE SoC fitting up to six cores and several neural network accelerators.

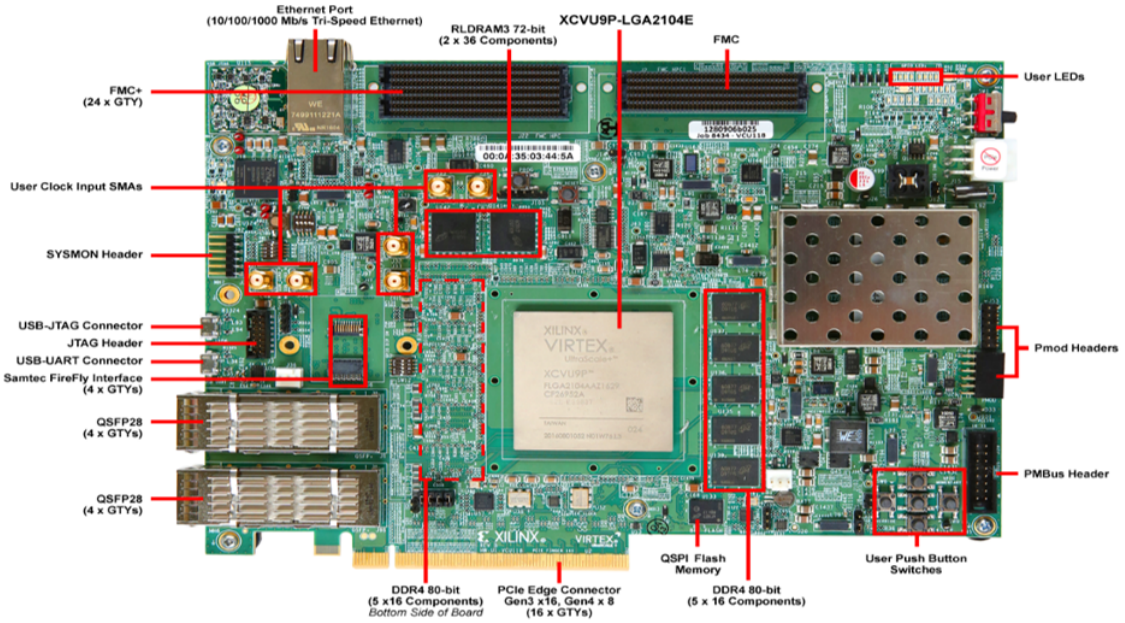

Figure 5.1: XILINX VCU 118 FPGA overview

This FPGA is shown in Figure 5.1. To allow using GRMON debugger, the configuration shown in 5.2 is needed.

Figure 5.2: Switch 12 configuration for UART debug link

For using the UART debug the switch 12 needs to be set as shown on Figure 5.2, then the micro-USB to UART interface needs to be connected on connector J4 (Figure 5.1). Then Ethernet needs to be set up by connecting the boards Ethernet connector J10 (Figure 5.1) to the PC through a network switch. If connected directly to the PC, some additional configuration is needed.

UART or JTAG debug links need to be properly setup in order for the Ethernet interface to work, as at each bitstream-flashing, a small piece of code configuring the Ethernet interface needs to be uploaded through this debug links to the board.

5.2 Benchmarks

In this section, a brief introduction to the benchmark suite used in our work is provided. As representative workload for critical control tasks, we use the CoreMark-pro benchmark suite. To resemble more performance demanding applications and to stress the shared resources as much as possible, we have designed several microbenchmarks that will be mapped to the secondary cores.

5.2.1. Coremark-PRO benchmark suite

"CoreMark®-PRO is a comprehensive, advanced processor benchmark that works with and enhances the market-proven industry-standard EEMBC CoreMark® benchmark. While CoreMark stresses the CPU pipeline, CoreMark-pro tests the entire processor." [24]

This is the benchmark that we chose to test the SELENE SoC and run as our main benchmark for this work.

On the test performed, we will be running different workloads of the CoreMark®-PRO benchmark on core zero and running several microbenchmarks on cores two, three and four.

The CoreMark®-PRO benchmarks are the following:

- cjpeg-rose7-preset: JPEG compression using 7 workers
- core: CoreMark
- linear_alg-mid-100x100-sp: Gaussian elimination with partial pivoting on a 100x100 single-precision matrix dataset
- loops-all-mid-10k-sp: Livermore Loops kernel with a 100x100 single-precision matrix
- nnet_test: Neural net simulation
- parser-12k: XML parser with a 125k dataset
- radix2-big-64k: FFT base-2 transform with 64k dataset
- sha-test: SHA256 benchmark
- zip-test: ZIP compression benchmark

To use the CoreMark®-PRO benchmark suite with the SELENE SoC, and the PMU, we reduced the number of iterations of the benchmarks to reduce the execution time in our system. This was done to be able to measure the execution time with the PMU counters

as they are only 32-bit wide and would overflow with the default number of iterations otherwise. Additionally, in the context of safety-critical control applications, tasks are usually periodically executed and the WCET derivation is usually relevant for the single activation of this task.

The number of iterations used for this work is:

Benchmark	Orginal number of iterations	New number of iterations
cjpeg-rose7-preset	10	1
core	1	1
radix2-big-64k	1000	10
sha-test	10	1
nnet_test	10	1
loops-all-mid-10k-sp	50	1
parser-125k	1	1
zip-test	1	1
linear_alg-mid-100x100-sp	50	1

Table 5.1: Number of iterations per benchmark for use with the PMU

5.2.2. Microbenchmarks

As for the microbenchmarks, we developed three C kernel functions that were compiled with the "-O0" gcc option. This option disables all optimizations and is necessary as to avoid compiler optimizations simplifying our benchmark.

Write microbenchmark

```
1 while(1){
2     volatile int i = 0;
3 }
```

The write microbenchmark presented above exploits the write-through nature of the L1 cache to force writes to memory. These writes to memory will pass through the L1, then be arbitrated on the AHB bus and pass to L2 or main memory, depending on the configuration. Note that the L2 can be activated or de-activated.

Read microbenchmark

```
1 volatile char* a = (char *) 0x00000000;
2 a = a+(coreId*32);
3 while(1){
4     *a;
5     a = (int)(a+(32*4))%((int) 0x40000000);
6 }
```

This read microbenchmark is slightly more complicated than the write microbenchmark. It exploits the capacity of the L2 cache to generate misses arbitrarily.

It does generate these misses by reading with the data on line 4. This line 4 generates a miss on every core thanks to the stride calculated on line 5. The stride that is applied is 32*4, being 32 the cache line size on bytes and 4 the max number of cores that will be running this microbenchmark concurrently. Finally, it applies a modulus calculation to the address as to not overflow the address range of the RAM.

Read-write microbenchmark

```
volatile int iterationCounter[4];
iterationCounter[coreId] = 0;
while(1){
    iterationCounter[coreId]++;
}
```

This simple microbenchmark creates a counter on main memory (iterationCounter), assigns it a zero at its beginning and increments its value continuously. Notice that this value is read from memory, not from the L1 cache, due to the iterationCounter variable implementing the volatile keyword.

This benchmark, thus, is a read-write benchmark due to the value of iteration counter being read from memory and written back to it. This benchmark proved useful to obtain the progress of each core on their workload and to ensure fairness between cores.

5.3 Performance evaluation of the improved static arbitration policy

5.3.1. Testing methodology

For the results, we experimented with the available control interfaces (a.k.a debug links) to find out which one causes less contention on the AHB bus. As we are testing on real hardware instead of simulation, control interfaces to communicate with the Gaisler GRMON debug interface to the FPGA are needed.

After experimentation, we determined that UART links caused a much heavier interference on the bus, so we decided to obtain the presented results using the Ethernet Interface as a debug link. This interference might be the result of UART being a slower interface and taking more time to transfer the same data since it used a narrower end link. We obtained the results using the PMU counters, specially the PMU cycle counter.

With this setup, we have only executed the tests once, as there is no operating system and thus, run to run standard deviation is minimal. This was decided after a study on run to run standard deviation with six samples of the same benchmark. This study concluded with a coefficient of variation of 0.0004%. This means that the standard deviation is 0.0004% of the mean between the measured execution times. That is, little execution time variability is expected in this type of experiments.

5.3.2. Performance results

Figure 5.3 represents the increase on execution time that the primary core incurs when sharing the system with different secondary core workloads.

As seen on Figure 5.3 the write contention microbenchmarks cause a much higher contention on average than the read benchmarks. This greater contention caused by the secondary cores ends up increasing the slowdown that the primary core will incur when using the write microbenchmark on the secondary cores.

This greater contention is due to writes allowing the progress of the underlying microbenchmark and thus, making it cause a greater contention. This would produce the pathological case of the benchmark performing reads and writes back to back and increasing the bus pressure substantially. An example of this behaviour can be seen at Figure 5.5.

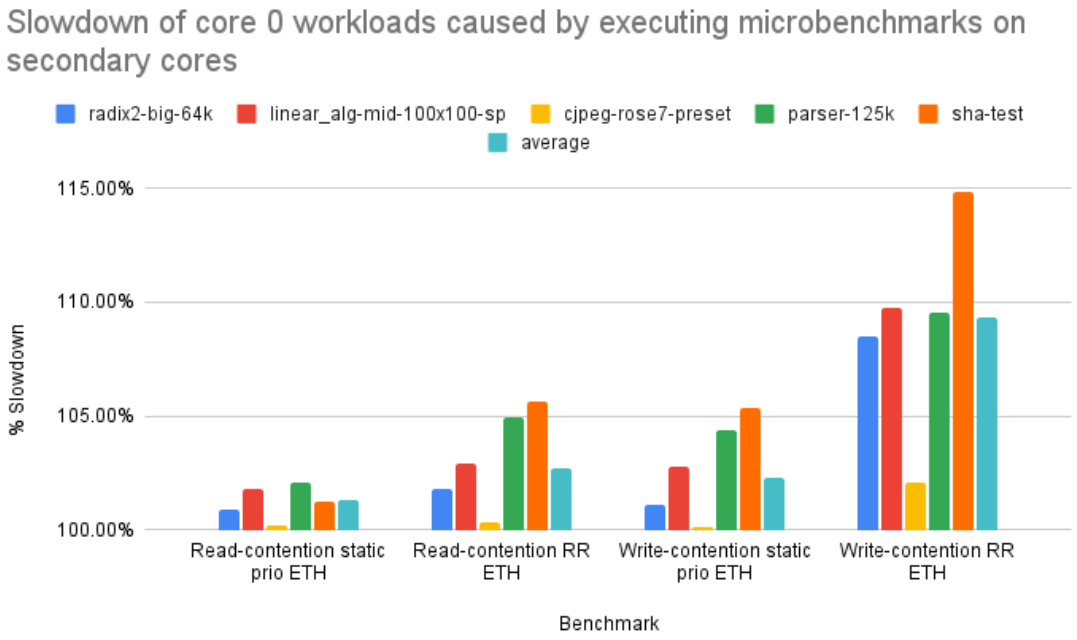

Figure 5.3: Slowdown of primary task incurred by sharing the bus depending on secondary core microbenchmark and primary core benchmark

For interpreting this schema we will make the following assumptions:

- No burst transactions are executed, so data transfer is just one cycle long.
- Read transactions have 2 cycles of latency and 1 cycles of data transmission due to not having bursts.
- Write transactions have 0 cycle of latency to execute. Furthermore, they take 1 cycle to make the data transmission as there is no burst transactions on this example.

These assumptions are realistic as writes are stored on a write buffer and don't block the cores progress while reads block the AHB bus until they are resolved. Furthermore, reads tend to take n cycles to solve, this cycles are caused due to cache seek times and memory access latency. No bursts are generated for accesses to small data structures.

In reality, access times are not constant as they vary according to the hierarchy level where the data is located. A realistic example would be having 1 cycle of access time to L1, 8 cycles to L2 and 18 cycles to main memory. However on this example, a constant 2 cycles is used.

As seen on Figure 5.5 there are two bus phases clearly distinguishable, the arbitration phase and the address/data phase. The arbitration phase is represented by the arbiter, giving the arbitration round to the core shown on its signals text. The arbitration and address/data phase are pipelined. This means that while the data is being transferred, the arbiter is selecting the master that should have the bus once the transfer is finished so back to back transfers are feasible.

The greater contention that we observed when using the write benchmark might be influenced by the effect shown on cycles 2-6 where the core 1 has a write that performs almost immediately, proceeding then to issue a read and increasing the probabilities of interfering with core 0. This is due to writes being non-blocking for the core, with reads being blocking due to the non-speculative nature of the SELENE NOEL-V cores.

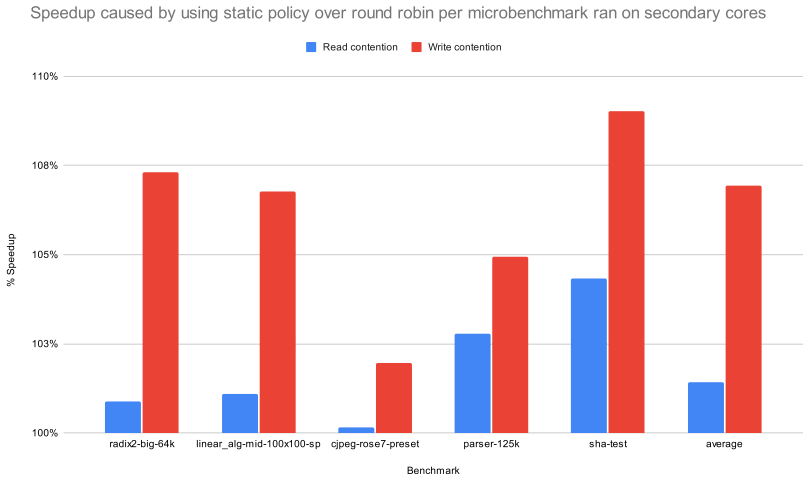

Figure 5.4: Difference in execution time per arbitration policy

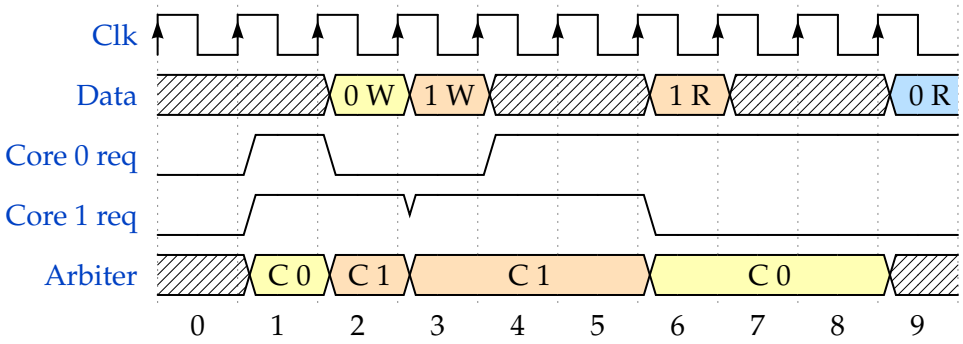

Figure 5.5: AHB bus read and write arbitration

This example helps to illustrate the limited effect of priority policies, where the priority master is blocked by the non-priority slave that currently has the bus, as seen from cycles 4-6 on the Figure 5.5. This effect is unavoidable and can present itself on a pathological manner. That is, core 1 systematically blocking core 0 by 2 cycles and always winning arbitration rounds on a system with static arbitration policies.

A solution to mitigate this pathological blocking effect is presented with the quota mechanism (section 4.2) and its quota refill (subsection 4.3.1) and randomized quota refill (subsection 4.3.2) improvements further bounding the maximum contention possible.

This blocking effect might be one of the causes of the different discrepancies between the benchmarks presented on Figure 5.3 and Figure 5.4.

Figure 5.4 presents the speedup of the primary core workload execution time that is obtained on our system when using the developed static arbitration mechanism over the round robin one.

One of the main possible affected benchmarks is sha-test, where the arbitration policy seems to have the most effect due to the memory dependence of this task. It is a great example to showcase the differences between static and Round Robin policies. Netting the static policy a 4% performance increase on the studied benchmark with the read microbenchmark and obtaining a 9% performance increase with the write microbenchmark as pictured on Figure 5.4.

On average, using the static arbitration policy that prioritizes the access to the bus of the critical core with the read microbenchmarks running in the contending cores renders

a 1.41 % increase on performance over the Round Robin policy and 6.94% with the write microbenchmark running in the contending cores. Thus, showing the effect described with Figure 5.5.

5.3.3. Summary

With the implemented static arbitration policy an improvement on critical task execution time of around a 7% is possible. The static arbitration policy does a best effort of prioritizing the primary core workload while maintaining the throughput of the system high (as no preemptive bus emptying is performed).

Static arbitration policy is preferred on systems where one core workload needs to be prioritized over the other core workload without the need of offering any performance guarantees. This static policy is simple and keeps the throughput of the system high.

5.4 Quota based arbitration

5.4.1. Testing methodology

To evaluate the quota based arbitration mechanism we will employ the read_write microbenchmarks. With this microbenchmarks we obtain a new metric on the secondary cores, the progress metric.

With this metric we create units of progress on secondary cores, being considered a unit of progress to read a value from memory, operate on it and write it back to memory. The more progress is achieved on the secondary cores while causing less contention to the main core, the better.

5.4.2. Baseline hardware quota implementation

To showcase the quota mechanism we will take a look at a simple example with low quotas and with the read write microbenchmark 5.2.2 running on all four cores.

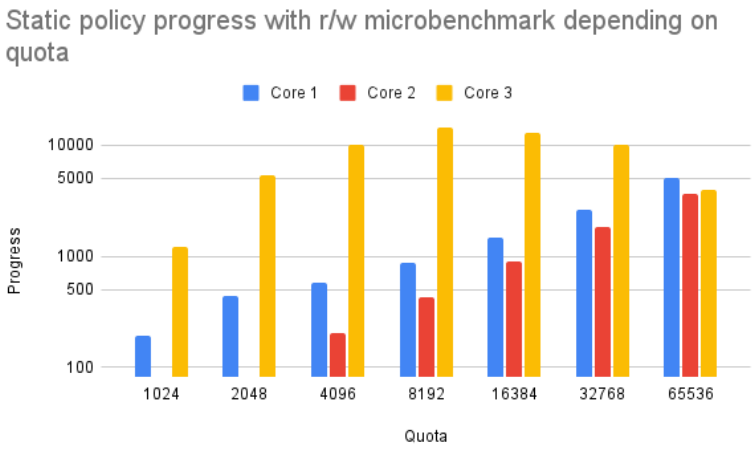

Figure 5.6: Read write microbenchmark secondary core progress depending on quota, static scheduling

The results shown on Figure 5.6, Figure 5.7 and Figure 5.8 are obtained running the read write microbenchmark until the core 0 reaches 10000000 read and writes. Then, the benchmark is stopped and the memory value of the read-write counters of cores 1, 2 and 3 is read from memory. The L2 cache is disabled on all benchmarks as it does not add any value in this setup and would pollute our results. The values read and written from memory are volatile, so there is no L1 cache interference.

Figure 5.6 shows the progress of the secondary cores for a given primary core workload and for varying quotas. This progress metric is bounded by the quota given to each core. This quota is decreased once the given core causes contention to the main core, that is also running the benchmark

By glancing at the results shown on Figure 5.6, we can observe the effect of pathological cases on each core progress until quota exhaustion. We can observe that, even though the static scheduling policy should prioritize the progress of core 1 over core 3, on the lower quotas the core 3 progress is heavily prioritized over the core 1 progress.

This priority inversion is the result of the static policy, where it seems that the priority cores are selected during a period where core 0 is specially busy, and thus, are using up their quota earlier. While, core 3, by not being prioritized during this busy period manages to make more progress by using up the same amount of core 0 contention than the other two cores.

But, finally, on the 65536 quota value data point the priorities get inverted back due to the high core 0 traffic period being lower than the used up core 0 quota. In fact, we can observe how, as all core quota increases from 8192 to 16384 the progress made by core 3 (yellow bar) decreases, even though its quota increased. This is due to the pathological case described before and is one of the reasons why we proposed the randomization mechanism that will be evaluated in the next section.

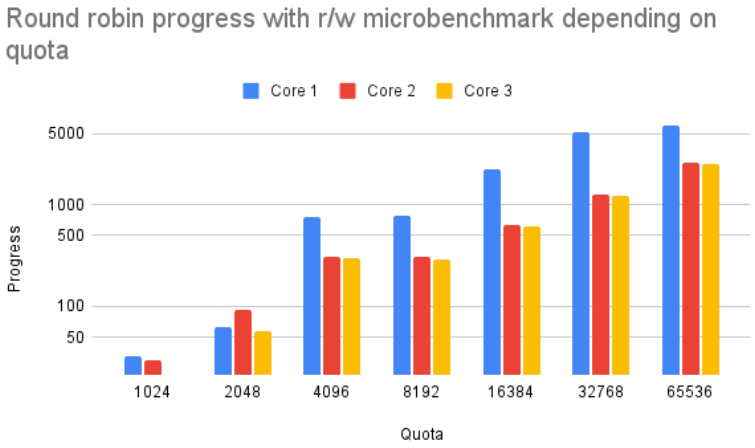

Figure 5.7: Read write microbenchmark secondary core progress depending on quota, Round robin scheduling

In Figure 5.7, we can observe how the progress with the round robin arbitration is much different than with the static arbitration. Here, it seems that core 1 progresses more than its peers with the same quota, this, even though it might seem beneficial is not really what we want with the round robin policy.

It is also worth noting that, even though we observe this behaviour on this particular examples it just serves to prove that, on this particular case, there are unexpected traffic patterns resulting from pathological cases on the bus.

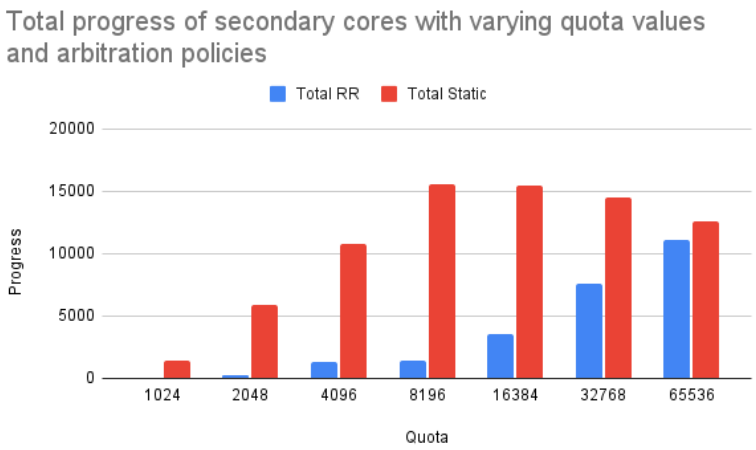

Figure 5.8: Total progress of secondary cores with varying quota values and arbitration policies

Finally, to complete this example we compare the total secondary core throughput with different quotas on Figure 5.8. This metric is obtained by making the sum between all the progress on the different secondary cores for each policy.

With this metric we can observe the need for more fine-grained control of the quota, as, in some cases, higher quota leads to different traffic patterns that resulted in an overall lower throughput of the secondary cores, and in turn, with a much higher contention caused to the main core. This negative traffic patterns can be seen on the static policy from the 8196 per core quota onward, where, when the total throughput should be higher the higher the contention we allow, but it becomes lower.

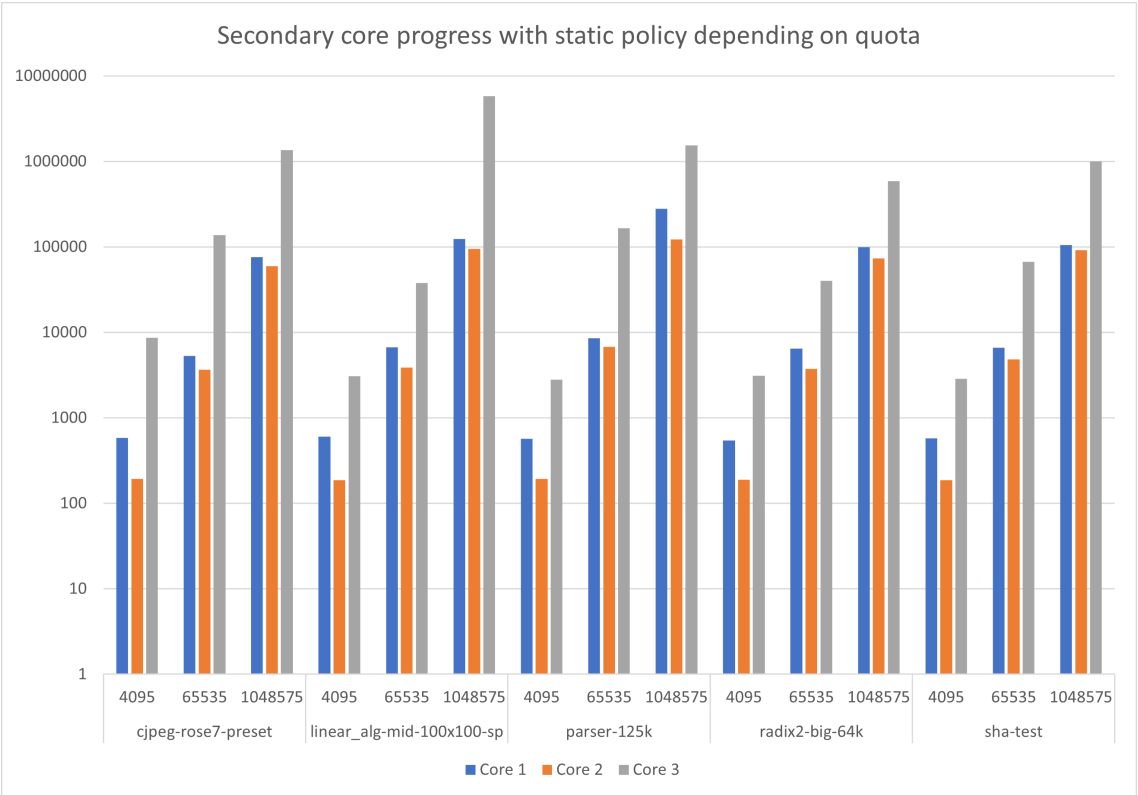

Figure 5.9: Read write microbenchmark secondary core progress depending on quota, static scheduling

Finally, Figure 5.9 shows the progress of the read and write microbenchmark on the secondary cores while running the EMBCC benchmarks on the main core for different quota contention allowances for the secondary cores.

It needs to be mentioned, that even though we applied the same quota to all secondary cores, our system is implemented in a way that lets us configure independent quotas and refill periods for each core.

Figure 5.9 proves that the phenomenon seen with the static arbitration policy and the non-refilling quota is pathological with all primary workloads. Figure 5.9 shows that for a non refilling quota, the read write microbenchmark and the static arbitration policy, the progress of core 3 is systematically greater than the on of core 1 and 2.

This is due to core 3 only being let into the bus when very low contention is present, and thus, when the probability of it blocking the core 0 workload is lower. This priority inversion phenomena is not ideal and will be mitigated with further improvements over the quota mechanism.

5.4.3. Summary

The hardware quota mechanism (explained on section 4.2 and which results are shown on the above section) lets us bound the maximum contention that secondary cores workloads can cause to the main workload while doing a best effort to preserve the performance of this secondary workloads.

But this result section presented us with the problem of troublesome bus patterns that end up consuming all available quota while performing little progress. This bus patterns can present themselves pathologically or punctually and will affect the performance of secondary core workloads while using the quota mechanism.

Overall, our implemented quota mechanism is ideal for guaranteeing a maximum contention to the primary core workload and bounding its WCET in a simple manner. This quota mechanism lets us bound WCET on a simple, software configurable manner, but, for cases where secondary core performance is important this mechanism is not sufficient.

5.5 The quota refill mechanism

In this section we will explore the effect of the quota refill mechanism explained on subsection 4.3.1. This quota refill mechanism is useful to rate-limit contending cores when they are causing a great effect to the progress of the critical core, but let them run at full speed when they are not.

This refill mechanism is implemented in a way that tries to decouple the cores each refill period. This decoupling is used to prevent the synchronization effect and is implemented by waiting till a quota period is met to refill it.

By construction, since the bus serializes requests to the shared resources, exhausting quota periods between different cores occurs at different times, so, in a way, we are forcing a decoupling of tasks that weakly randomizes the refill of quotas. Actual randomization will be explored on further subsections.

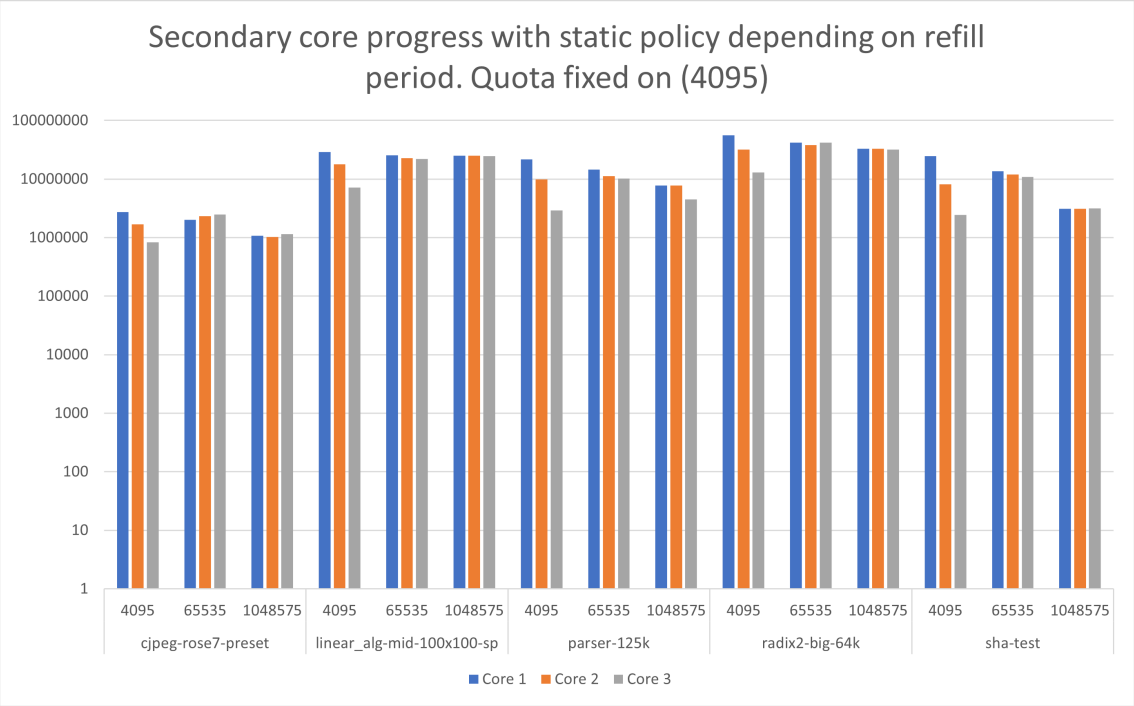

Figure 5.10: Progress on secondary cores with different refill periods and static policy

Figure 5.10 presents the same testing methodology and benchmarks as Figure 5.9, but varying the quota refill period instead of the quota variable. The selected quota is 4095. With this quota we observed large discrepancies on all benchmarks on Figure 5.9 and core 3 having an advantage systematically.

Conversely, when we apply the quota refill mechanism as seen on Figure 5.10 we observe the results that we initially expected from a static arbitration policy. In this case, and specially with lower refill periods (Higher allowed contention) the core 1 workload progresses substantially more than the core 3 workload, as is the one that has the highest priority.

This progress of the core 0 workload comes at a cost of higher contention to the main core. As the refill period is not constant and is only applied at quota exhaustion of each individual core, and core 0 has the highest priority it exhausts its quota substantially faster than core 3. This causes that, for the majority of the workloads, core 1 generates more contention to the main core than core 3 and core 3 is starved over the majority of the time.

Core 1 generating more contention than core 3 with the static policy is expected, specially with the lower refill periods. This higher contention generation can be as high as 3.3 times more contention generated by core 1 on sha-test than core 3.

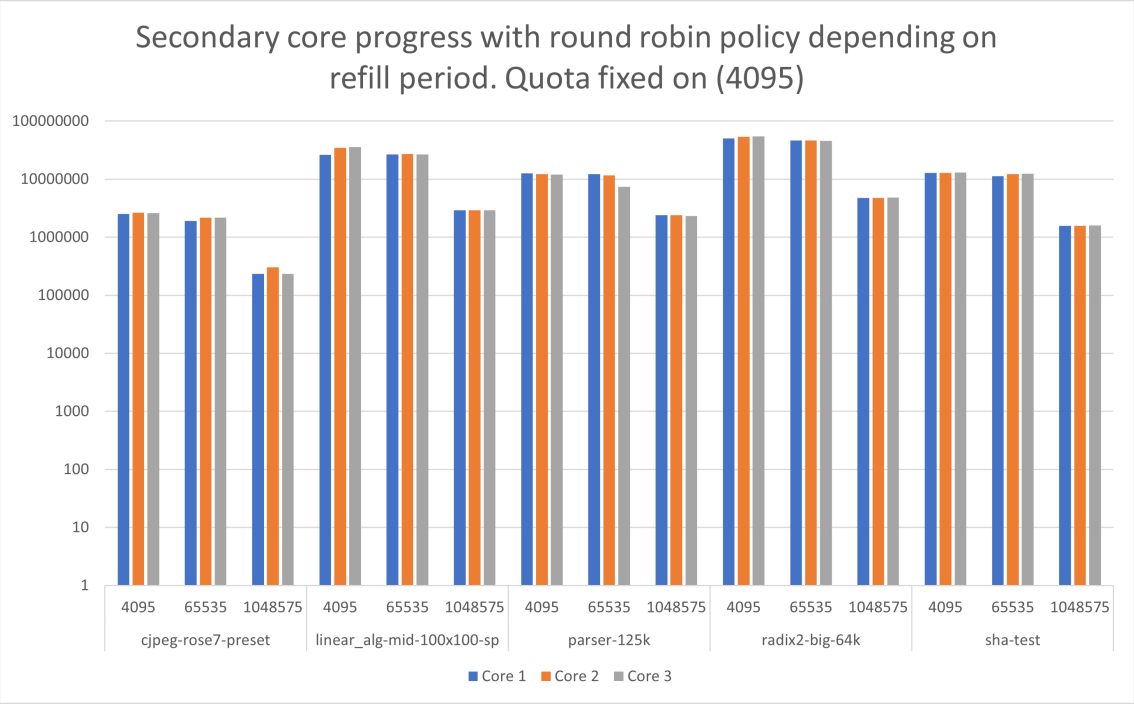

Figure 5.11: Progress on secondary cores with different refill periods and round robin policy

This low refill period effect of acting as a regulator for our scheduling policy can also be seen with the round robin results shown on Figure 5.11, where for the lower refill periods the results are closer together than for the higher refill periods.

5.5.1. Randomized quota refill

As explained before, the quota refill mechanism introduces some level of randomization with the rate limiting period being executed on quota exhaustion. However, this small randomization cannot ensure that the system does not present systematic pathological situations in which the critical task is blocked by another non-critical one.

To ensure that our critical task is not systematically blocked, we introduce the randomization of refill periods, from now on all results will use this method as is the only one that can guarantee the alignment of requests if free from systematic pathological alignment. Without this quota randomization we cannot ensure that one edge case will not generate a traffic pattern that consistently blocks the critical core and increases our critical workload runtime substantially, which in turn penalizes the progress that this non-critical task can make due to exhausting the assigned quota very frequently.

In general, the results obtained applying randomization are similar to those obtained without the explicit randomization and presented on Figure 5.11 and Figure 5.10 so we will not be showing the secondary workload progress with the EMBC benchmarks on the main one. These results are the same as the randomization period does not improve execution time average, it is just useful to ensure homogeneity between runs.

Now with this higher quality data we can explore the effect of different quota refill periods with different workloads.

By leveraging the quota refill mechanism on soft real time or less critical tasks and carefully knowing which workload is running on each core, we can tune the quota and quota refill period parameter to obtain the best possible efficiency out of our system. On Figure 5.12 a first approximation at this work is presented.

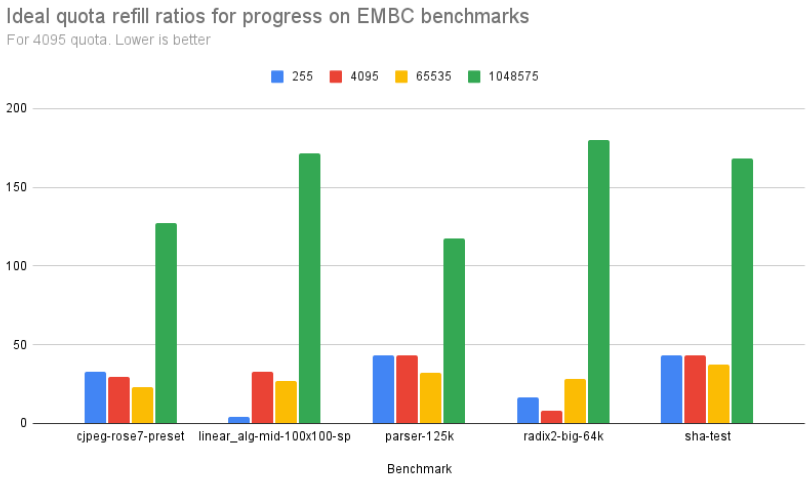

Figure 5.12: Ideal quota and refill ratios for EMBC benchmarks on the main core and read write microbenchmark on the secondary cores

Figure 5.12 presents a new metric for this work, this metric compares the cycles taken to execute the main core workload and the sum of all secondary workload progresses. By combining this two metrics we obtain the one shown on Figure 5.12, this metric measures the amount of processor cycles that will take for any secondary core to progress. With this metric, the lower the value, the more efficient is the configuration. The lower the value, the less cycles of interference to the main core each unit of progress of the secondary cores causes.

$$cyclesPerProgress = \frac{TotalExecutionTimeMainWorkload(cycles)}{Core1Work + Core2Work + Core3Work}$$

By observing the cyclesPerProgress metric on Figure 5.12 we can observe that, for cjpeg-rose7-preset the ideal quota-refill ratio is around 4095 for quota and 65535 for refill period. While for radix, the ideal refill period for the aforementioned quota seems to be around 4095 cycles and for linear_alg_mid-100x100-sp it seems to be of around 255. The goal of this mechanism is leveraging the refill mechanism to strike a balance between critical core progress and secondary core progress.

5.5.2. Summary

By using the quota refill mechanism (Explained originally on subsection 4.3.1) with the correct refill periods and with randomization activated we can break the priority inversion effect shown on the previous section. This quota refill mechanism greatly improves the performance of the secondary cores and manages to break the majority of pathological cases over a large period of time.

Using the refill period randomization we can ensure to break all systematic pathological cases and using the leaky bucket mechanism we can get more fine grained control over the maximum allowed quota of the secondary cores.

Using this quota refill mechanism we can also find that, for different traffic patterns of different primary workload applications, different quota and refill ratios are optimal. This optimal refill ratios are a result of the non-saturation of the bus while maintaining performance guarantees on the main core and can be studied and tuned for each specific

application. Thus, that mechanism is well suited for applications where more than one core requires predictable timing.

Additionally, we encourage to use the quota refill/leaky bucket mechanism as a way to mitigate all pathological bus patterns and guarantee a bound to the maximum possible contention on each workload while doing a best effort to ensure progress on secondary cores.

CHAPTER 6
Conclusions

In this project, our goal has been improving the performance guarantees of critical workloads on mixed-criticality systems. In order to do this, we first observed the possible approaches to implement a mixed-criticality system. From total partitioning of resources, to no partitioning of resources, and assuming the worst possible interference in the critical core workload, to a mix of guaranteeing some progress to the critical workload by limiting its possible interference.

We have managed to limit the maximum possible interference to the critical workload by using a mix of custom arbitration policies and a quota mechanism that rate-limits the contending core requests once an interference threshold is reached. This new family of solutions that have been implemented in the SELENE SoC offer new optimization opportunities to a wider range of applications beyond simply stopping the progress of less critical tasks.

By leveraging the PMU and interconnecting it with the shared bus arbiter on the SELENE SoC, we achieved the objective of bounding he maximum possible interference time on the shared bus for our workload using a software configurable quota mechanism which can guarantee the derivation of a safe WCET estimation. Additionally, with this software configurable quota mechanism, a best effort is made to achieve the maximum possible progress on the secondary cores while keeping the WCET of the primary core.

We also improved over the quota mechanism by implementing a self refilling quota that offers performance guarantees for less stringent real-time applications and a leaky bucket mechanism that offers performance guarantees on hard real-time systems. At the same time no software intervention is needed to reset the quotas.

With this leaky bucket mechanism we can control the allowed interference to the main workload progress, thus, simplifying WCET calculus while doing a best effort to not starve secondary core workloads.

Finally, we implemented a refill period randomization mechanism that lets us be able to avoid pathological cases on our system, thus, eliminating the previously observed AHB synchrony effect observed on [4, p. 4] and thus, increasing the performance predictability of our system.

Overall, we implemented a highly customizable family of shared resources arbitration schemes that can offer a wide variety of levels of quality of service for mixed criticality systems.

Bibliography

[1] *ISO 26262 - Road Vehicles Functional Safety*. URL: https://www.iso.org/standard/68383.html (visited on 2021-09-02).

[2] Christopher B. Watkins and Randy Walter. "Transitioning from federated avionics architectures to Integrated Modular Avionics". In: *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*. 2007, 2.A.1-1-2.A.1–10. DOI: 10.1109/DASC.2007.4391842.

[3] Selma Saidi, Sebastian Steinhorst, Arne Hamann, et al. "Special Session: Future Automotive Systems Design: Research Challenges and Opportunities". In: *2018 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 2018, pp. 1–7. DOI: 10.1109/CODESISSS.2018.8525873.

[4] Gabriel Fernandez, Javier Jalle, Jaume Abella, et al. "Computing Safe Contention Bounds for Multicore Resources with Round-Robin and FIFO Arbitration". In: *IEEE Transactions on Computers* 66.4 (2017), pp. 586–600. DOI: 10.1109/TC.2016.2616307.

[5] Alfons Crespo, Alejandro Alonso, Marga Marcos, et al. "Mixed Criticality in Control Systems". In: *IFAC Proceedings Volumes* 47.3 (2014). 19th IFAC World Congress, pp. 12261–12271. ISSN: 1474-6670. DOI: https://doi.org/10.3182/20140824-6-ZA-1003.02004. URL: https://www.sciencedirect.com/science/article/pii/S1474667016435664.

[6] Javier Jalle, Leonidas Kosmidis, Jaume Abella, et al. "Bus designs for time-probabilistic multicore processors". In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2014, pp. 1–6. DOI: 10.7873/DATE.2014.063.

[7] C. Park and A. Shaw. "Experiments with a program timing tool based on source-level timing schema". In: *Computer* 24 (1991), pp. 48–57.

[8] Thomas Lundqvist and Per Stenstrom. "An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution". In: *Real-Time Systems* 17 (Nov. 1999), pp. 183–207. DOI: 10.1023/A:1008138407139.

[9] Guillem Bernat, Antoine Colin, and Stefan Petters. "pWCET: a Tool for Probabilistic Worst-Case Execution Time Analysis of Real-Time Systems". In: (June 2003).

[10] Tomás Picornell, José Flich, Carles Hernández, et al. "Enforcing Predictability of Many-Cores With DCFNoC". In: *IEEE Transactions on Computers* 70.2 (2021), pp. 270–283. DOI: 10.1109/TC.2020.2987797.

[11] Jieming Yin, Pingqiang Zhou, Sachin S. Sapatnekar, et al. "Energy-Efficient Time-Division Multiplexed Hybrid-Switched NoC for Heterogeneous Multicore Systems". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 2014, pp. 293–303. DOI: 10.1109/IPDPS.2014.40.

[12] Tomás Picornell Sanjuan, José Flich, Carles Hernández, et al. "Enforcing Predictability of Many-cores with DCFNoC". In: *IEEE Transactions on Computers* PP (Apr. 2020), pp. 1–1. DOI: 10.1109/TC.2020.2987797.

[13] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, et al. "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms". In: *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2013, pp. 55–64. DOI: 10.1109/RTAS.2013.6531079.

[14] Jan Nowotsch, Michael Paulitsch, Daniel Bühler, et al. "Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement". In: *2014 26th Euromicro Conference on Real-Time Systems*. 2014, pp. 109–118. DOI: 10.1109/ECRTS.2014.20.

[15] *Arm AMBA 5 AHB Protocol Specification*. 5th ed. ARM. Oct. 2015.

[16] *AHB Example AMBA SYstem*. 1st ed. ARM. Jan. 1999.

[17] Andreas Kurth, Wolfgang Rönninger, Thomas Benz, et al. "An Open-Source Platform for High-Performance Non-Coherent On-Chip Communication". In: *CoRR* abs/2009.05334 (2020). arXiv: 2009.05334. URL: https://arxiv.org/abs/2009.05334.

[18] *Selene H2020 webpage*. URL: https://www.selene-project.eu/project-overview/project (visited on 2021-06-27).

[19] *Micron MT40A256M16GE-083E IT memory model*. URL: https://www.micron.com/products/dram/ddr4-sdram/part-catalog/mt40a256m16ge-083e-it (visited on 2021-07-28).

[20] Gaisler. *Noel-v*. URL: https://www.gaisler.com/index.php/products/processors/noel-v (visited on 2021-06-22).

[21] *SafePMU User's Manual*. BSC. Dec. 2020.

[22] *GRLIB IP Core User's Manual*. Cobham Gaisler AB. Apr. 2021.

[23] *GRMON3 User's Manual*. Cobham Gaisler AB. June 2021.

[24] Embedded microprocessor benchmark consortium. *CoreMark-pro. An EEMBC Benchmark*. URL: https://www.eembc.org/coremark-pro/ (visited on 2021-06-25).

[25] *GNU Make*. URL: https://www.gnu.org/software/make (visited on 2021-09-03).

APPENDIX A
Tools used and relationship between the studied masters degree to this work

A.1 Tools used

As for the tools used, we used many standard hardware implementation and verification tools, as well as scripting languages and simulation frameworks.

A.1.1. Xilinx Vivado

For synthetising the Selene SoC design we used Vivado on its CLI (Command Line Interface) mode.

With this mode of Vivado there was no need to use the graphical user interface and thus, efficient scripts could be written. The whole syntetization process is based on "GNU make" and Vivado TCL scripts.

In order to use our target VCU118 board, the usage of Vivado to synthetize the bitstream to be uploaded is mandatory as there is no other tool that can syntetize bitstreams for Xilinx FPGA boards.

We also extensively used the Vivado library and IP-core suite to implement our memory controller and some core functionality of our SoC, such as the accelerators, created with Vivado HLS and ported into our design with Vivado.

A.1.2. Mentor Graphics Questasim

For simulating the Selene SoC and our proposed implementations we used the Questasim simulator. This simulator offers great tools for debugging by offering step by step debugging for Verilog processes and mixed SV (System verilog) and VHDL simulation.

We extensively used the memory snooping capabilities, the waveform tool and the breakpoints tool. The majority of complex mechanisms that we implemented on the SELENE SoC where designed using the Questasim to debug and verify.

We extensively used the Questasim IDE to edit and debug our code, and the Questasim waveform tool debug it and ensure the system has the expected behaviour. We also made use of the TCL scripting language inside the Questasim tool to ease the debugging process.

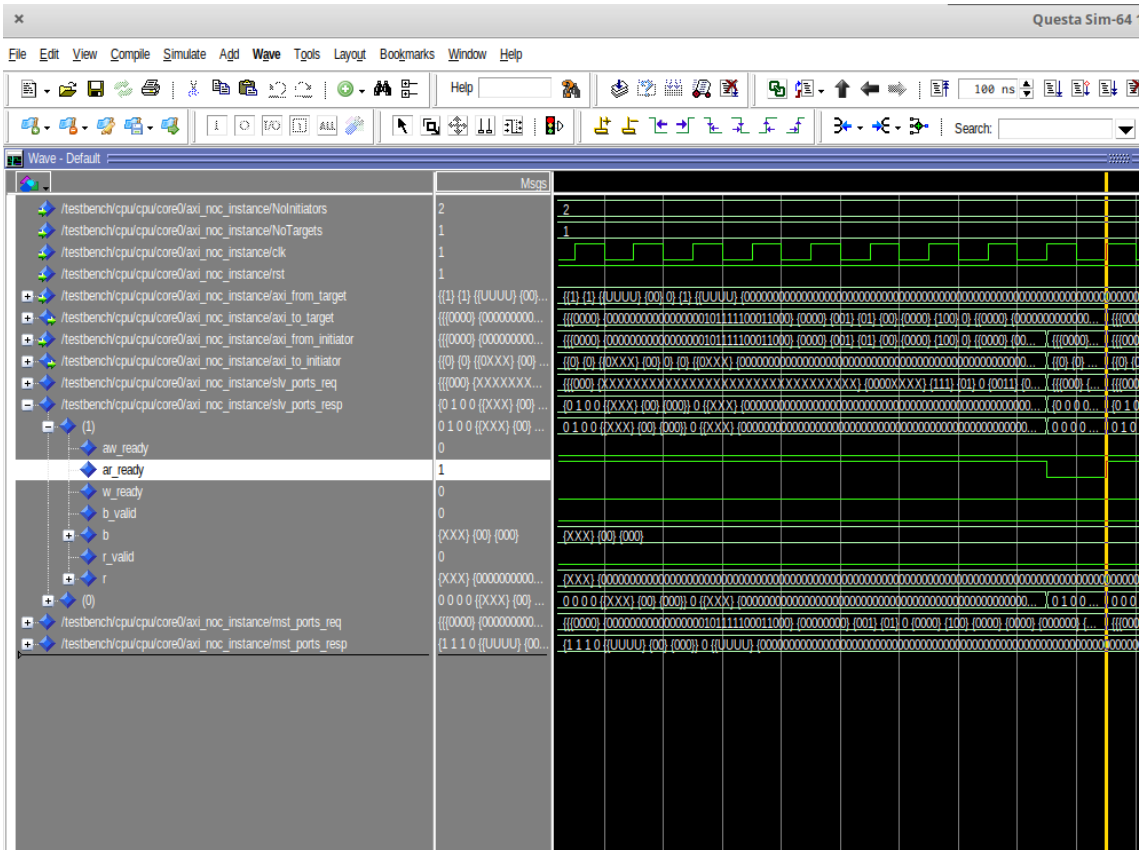

Figure A.1: Questasim waveform screenshot

A.1.3. GNU Make

> Make gets its knowledge of how to build your program from a file called the makefile, which lists each of the non-source files and how to compute it from other files. When you write a program, you should write a makefile for it, so that it is possible to use Make to build and install the program.[25]

The GNU Make tool is an indispensable tool to automate compilation workloads inside a big project such as the SELENE H2020 Project. We extensively used the Make tool to create a tree of make targets, integrating our VCU118 specific workflow with the existing generic Gaisler GRLIB make hierarchy.

A.1.4. Shell scripting, GRMON scripting and Python scripting

For the obtention and parsing of the majority of results of this work we used a combination of C software programs running on the SELENE SoC and shell scripts used to interface with GRMON and obtain the results of the programs running on the FPGA.

This project uses a wide variety of bash scripts, shell scripts, GRMON scripts and python scripts to obtain the results on a CSV (Comma separated value) manner. This csv files are necessary to easily import the results into Microsoft Excel and Google Sheets, the two graphing programs used on this work to obtain the presented graphs.

We used Python to further parse the results of reading the PMU memory addresses with GRMON, converting from hexadecimal to decimal the values and assigning the overflow bit correctly to each of the read results.

A.2 Relation between the studied maters degree and this work

There is a strong relation between the studied masters degree and this work. This masters degree is the computer architecture and networking masters degree. In this masters degree there is a strong focus on interconnection networks and computer architecture, specially at the microprocessor level.

The strong focus on interconnection networks of this master made it much easier to understand the concepts used on this work and the structure of the AMBA AHB standard. The REC (Redes en chip) *on chip networks* subject was specially relevant for this work, as we had a seminar directly orientated to the AMBA AHB standard.

This masters degree also focuses on designing small paper-like works with latex, giving us the tools and expertise to easily create this TFM on latex already knowing how to use this PDF creation language.

Finally, the studied masters degree also puts a strong emphasis on chip design and internal chip structure with its ATP subject. Where I understood the internal architecture of a complex processor and the importance of the memory subsystem, as well as the indeterminism that this memory subsystem can produce. This understanding of computer processors was also used on this work to better understand the traffic needs of the Selene NOEL-V cores.