UNIVERSITY OF MODENA AND REGGIO EMILIA

Department of Physics, Informatics and Mathematics
Ph.D degree in Mathematics
in agreement with the University of Ferrara and the University of Parma
Cycle XXXIV

DOCTORAL THESIS

Memory Interference and Mitigations in Reconfigurable HeSoCs for Embedded AI

Author:
Dott. Gianluca BRILLI

Supervisor:
Prof. Marko BERTOGNA

Co-Supervisor:
Dott. Ric. Paolo BURGIO

Ph.D. Course Coordinator: Prof. Cristian Giardinà

Academic Year 2019/2021

Declaration of Authorship

I, Dott. Gianluca BRILLI, declare that this thesis titled, "Memory Interference and Mitigations in Reconfigurable HeSoCs for Embedded AI" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date: 19/02/2022

UNIVERSITY OF MODENA AND REGGIO EMILIA

Abstract

Department of Physics, Informatics and Mathematics

Doctor of Philosophy

Memory Interference and Mitigations in Reconfigurable HeSoCs for Embedded AI

by Dott. Gianluca BRILLI

Artificial Intelligence (AI) is widely adopted in the embedded domain and especially in *Cyber Physical Systems (CPS)*. To match the computational requirements of modern CPS, chip manufacturers are adopting a heterogeneous design, where sequential processors and energy-efficient massively parallel accelerators are used to perform complex tasks in an energy efficient manner. *Commercial-Off-The-Shelf (COTS)* platforms are often adopted, where the memory hierarchy, composed of multiple cache layers and a main memory (*DRAM*), is shared between the computational engines of the system. This scheme allows to increase the time-to-market, the scalability of the system and can provide adequate average-case performance. However, a shared resources template may lead to unbounded delays due to contention [12, 43, 5]. This behaviour could compromise the adoption of COTS HeSoCs in critical applications, where the system must provide bounded performance even in the worst-case. *Predictable Execution Models (PREM)* [55, 46, 2, 40, 22], are proposed to solve DRAM contention on multicores by imposing mutual exclusion, a schema that does not allow to exploit the bandwidth capabilities of modern HeSoCs. This doctoral dissertation will cover three main aspects: *i)* an in-depth study on memory contention problem focusing on HeSoCs, especially FPGA-based; *ii)* a cycle-accurate methodology, based on memory bandwidth throttling, to overcome the PREM-like memory bandwidth under-utilization, while mitigating DRAM interference; *iii)* an evaluation of the proposed methodology considering real-world applications from the *embedded* domain: an FPGA-accelerated infrastructure to localize a car inside a map and one that aims to extensively characterize objects detection Neural Networks for autonomous systems perception.

Estratto

L'*Intelligenza Artificiale (IA)* è ampiamente adottata nel dominio embedded ed in particolare nei *Sistemi Cyber Fisici (CPS)*. Per far fronte ai requisiti di calcolo dei moderni CPS, i produttori di chip stanno adottando un design eterogeneo, in cui vengono affiancati ai tradizionali processori sequenziali, acceleratori paralleli per eseguire compiti complessi in modo efficiente. Le piattaforme *Commercial-Off-The-Shelf (COTS)* sono al giorno d'oggi preferite da sviluppatori e system designer, in quanto permettono un rapido inserimento di nuovi prodotti sul mercato. Tuttavia tali sistemi si basano su un modello architetturale a risorse condivise ed in particolare la gerarchia di memoria, composta da più livelli di cache ed una memoria principale (*DRAM*), è condivisa tra differenti componenti hardware all'interno del sistema. Un modello architetturale basato su risorse condivise può comportare incrementi non predicibili sul tempo di esecuzione dei task all'interno del sistema, a causa di problemi di contesa [12, 43, 5]. Questo comportamento potrebbe compromettere l'adozione di COTS HeSoC in applicazioni critiche, in cui il sistema deve fornire prestazioni garantite anche nel worst-case. *Predictable Execution Models (PREM)* [55, 46, 2, 40, 22], sono proposti per risolvere la contesa di memoria su sistemi multicore imponendo *mutual exclusion* sugli accessi in memoria, uno schema che non consente di sfruttare le elevate capacità di larghezza di banda dei moderni HeSoC. Questa tesi di dottorato si focalizza su tre aspetti principali: *i)* uno studio approfondito sul problema della contesa di memoria incentrato sugli HeSoC, in particolare basati su FPGA; *ii)* una metodologia che tramite un meccanismo di bandwidth throttling permette un'accurata regolazione della banda della memoria, al fine di risolvere il sotto-utilizzo della DRAM bandwidth degli approcci di tipo PREM, mitigando al contempo l'interferenza su DRAM condivisa; *iii)* una valutazione della metodologia proposta considerando applicazioni reali nel dominio *embedded*: un'infrastruttura accelerata su FPGA per la localizzazione di robot all'interno mappe pre-costruite ed una estensiva caratterizzazione di Reti neurali per rilevamento oggetti ampiamente utilizzate come sistemi di percezione in sistemi autonomi.

Acknowledgements

Questa tesi è frutto di un difficile percorso durato tre anni, che mi ha richiesto duro lavoro e dedizione, ma che mi ha dato la possibilità di lavorare ad appassionanti tematiche di ricerca. Al termine di questo percorso vorrei ringraziare diverse persone senza le quali questo traguardo finale non sarebbe stato possibile.

Vorrei perciò ringraziare **Marko Bertogna** per avermi accolto all'interno del suo gruppo di ricerca ed avermi dato la possibilità di lavorare in un ambiente molto positivo e amichevole. In particolar modo ci terrei a ringraziare **Paolo Burgio** e **Andrea Marongiu** per avermi guidato durante il dottorato, vi sono davvero grato per aver creduto in me. Estendo inoltre i ringraziamenti ad **Alessandro Capotondi** e **Paolo Valente** per avermi fornito preziosi consigli per migliorare il mio lavoro. Ringrazio inoltre i revisori **Paolo Meloni** e **Marco Domenico Santambrogio** per avermi fornito utili consigli per migliorare la mia tesi durante il processo di revisione.

Ringrazio inoltre tutti i miei colleghi dell'*HiPeRTLab* con i quali ho avuto il piacere di lavorare in questi tre anni. In particolar modo vorrei ringraziare i ragazzi della divisione FPGA del laboratorio: **Andrea Bernardi**, **Federico Gavioli**, **Michele Guzzinati** e **Gianluca Bellocchi** con i quali ho avuto modo di collaborare in maniera più stretta e scambiare idee e consigli. Vorrei inoltre ringraziare **Giacomo Valente**, ricercatore dell'Università dell'Aquila, col quale ho avuto modo di collaborare durante l'ultimo anno di dottorato ed imparare dalla sua esperienza in ambito embedded. Ringrazio inoltre **Elisa Landini** per aver avuto la pazienza di insegnarmi le basi per la gestione di un progetto europeo.

Estendo inoltre i ringraziamenti ai **miei amici** ed ai **miei genitori** per il sostegno che mi hanno sempre dato, in particolare durante gli anni di dottorato.

Contents

Declaration of Authorship iii

Abstract vi

Acknowledgements ix

1 Introduction 1

1.1 The heterogeneous systems revolution . . . 3

1.2 The other side of the coin . . . 4

1.3 Contributions of this research . . . 6

2 Overview of Embedded Heterogeneous Systems 9

2.1 General Template . . . 9

2.2 NVIDIA Tegra X2 . . . 11

2.3 NVIDIA Xavier AGX . . . 12

2.4 Xilinx Zynq UltraScale+ . . . 12

2.5 Xilinx Versal ACAP . . . 15

3 Related Work 17

3.1 Main topics . . . 17

3.2 Memory interference and platform characterization . . . 17

3.2.1 CPU and GPGPU platforms . . . 18

3.2.2 FPGA-based platforms . . . 18

3.3 Approaches to mitigate interference . . . 19

3.3.1 Software approaches . . . 19

3.3.2 Hardware extensions . . . 21

FPGA-based approaches and custom HW solutions . . . 21

Platform-specific solutions . . . 23

4 A study on Memory Interference in HeSoCs 25

4.1 Introduction . . . 25

4.2 Interference Analysis of Shared L2-cache on GPGPUs 26
4.2.1 Experimental Setup 28
4.2.2 Experimental Results 30
4.2.3 Conclusion 33
4.3 Interference Analysis of Shared Main Memory on FPGA-based HeSoCs 34
4.3.1 Accelerator Template with CMRI support 35
4.3.2 Experimental Results 38
Self interference among CPU cores 40
Memory Interference among heterogeneous engines 43
4.3.3 Conclusions 52

5 The Proposed Mechanism to Mitigate Memory Interference 55
5.1 Introduction 55
5.2 The Proposed Mechanism 57
5.2.1 Monitoring 57
5.2.2 Throttling 59
5.2.3 Example application 61
5.3 Experimental Results and SOA Comparison 64
5.3.1 Benefits of CMRI technique 65
Memory interference mitigation analysis 65
Improvements on bandwidth exploitation 67
5.3.2 Cost of HW and SW Monitoring and Throttling 68
5.3.3 Tightly-coupled versus Loosely-coupled monitoring and throttling 74
5.3.4 QoS for memory interference mitigation 78
Comparison with Zynq UltraScale+ QoS ecosystem 79
Real-world benchmarks 79
Synthetic benchmarks 80
5.4 Conclusion 83

6 Real-World Embedded Applications 85
6.1 Particle Filter localization 86
6.1.1 FPGA-Overlay for Real-Time Localization 86
6.1.2 Background 88
6.1.3 The System Architecture 88
6.1.4 Evaluation 90

6.1.5 Conclusion 92
6.2 CNN-based object Detection and Classification 92
6.2.1 Metrics of Interest 93
6.2.2 Neural Network Comparisons 94
6.2.3 Lesson Learned 95
6.2.4 optimizations 96
6.2.5 Conclusion 97

7 Conclusion **99**
7.1 Future research directions 100

8 List of Publications **103**

Bibliography **105**

List of Figures

1.1 General Cyber-Physical System, composed of the physical world and an Embedded System used as its central compute environment 2
1.2 Main memory bandwidth requests over time required by the Xilinx DPU accelerator. The neural network performs a *pose detection* task on Zynq UltraScale+ device. 5

2.1 General architectural template for heterogeneous SoCs, comprising both host cores and accelerators. 9
2.2 Block diagram of the NVIDIA Tegra X2. 11
2.3 Block diagram of the NVIDIA Xavier AGX. 13
2.4 Block diagram of the ZU9EG MPSoC, the chip within the Xilinx ZCU102 evaluation platform. 14
2.5 Block diagram of the VC1902 ACAP, the chip within the Xilinx VCK190 evaluation platform. 15

4.1 Reference iGPU architecture with key architectural bottlenecks 27
4.2 Thread access pattern to LLC. The case reported in this figure is a *coalesced* access, that occurs when the *stride* is equal to 1. In this case only the first thread causes a cache miss (the red cell). 29
4.3 Slowdown of the *vadd* kernel when executed concurrently with an interference kernel. 30
4.4 Slowdown of the *vadd* kernel when executed concurrently with a CUDA stream that performs memory copy *host-to-device*. 31
4.5 Slowdown of the *gemm* kernel when executed concurrently with an interference kernel. 31
4.6 Slowdown of the *gemm* kernel when executed concurrently with a CUDA stream that performs memory copy *host-to-device*. 32

4.7 Two architectural designs for accelerator deployment. On the left a more traditional design, where the accelerator and DMA datapaths are controlled by an FSM. On the right the architectural design used in this work, where the control is implemented by a soft-processor. . . 35
4.8 Tradeoff between fine-grained memory transactions and DMA programming overhead. 37
4.9 Impact of adopting fine-grained *subcopies*. 38
4.10 Accelerator Clusters 41
4.11 Comparison of the DRAM bandwidth available on the host cores of the two systems. (4.11a) the results obtained on R5F and (4.11b) the same test on A72. 42
4.12 Alternating Direction Implicit solver (ADI) 45
4.13 Matrix Transpose and Vector Multiplication (ATAX) 45
4.14 BiCG Sub Kernel of BiCGStab Linear Solver (BICG) 45
4.15 2-D Convolution (Conv2D) 46
4.16 Covariance Computation 46
4.17 2-D Finite Different Time Domain Kernel (FDTD-2D) 46
4.18 Cholesky Decomposition 47
4.19 Matrix Vector Product and Transpose (MVT) 47
4.20 Triangular solver (TriSolv) 47
4.21 Toeplitz system solver (Durbin) 48
4.22 Gram-Schmidt decomposition 48
4.23 LU decomposition 48
4.24 Sequential memory access pattern 50
4.25 Random memory access pattern 50
4.26 Convolutional Neural Networks on Versal AI Engine 52

5.1 FPGA-based HeSoC Accelerator Cluster Template. 56
5.2 The RBR internal signals. `state` is the enable signal, that goes low when the sniffer sends an `IDLE_valid`. `IDLE_CYCLES` is the output of the weigher IP, based on CLI and `COPY_CYCLES`. 58
5.3 The internal signals of the throttler component. The AXI4 handshake signals (xvalid/xready) are masked based on the `state` of the throttler. 61
5.4 Runtime Bandwidth Regulator (RBR). 63

5.5 *CMRI* on different Processing Elements and platforms. (5.5a and 5.5b): Polybench on APUs; (5.5c): CNNs on AI Engines; (5.5d): Micro-benchmarks on R5F. 66
5.6 Analysis of the memory bandwidth exploitation improvements compared to PREM-like schemes. The tasks under test run on *ARM Cortex A53* of the *Zynq UltraScale+ MPSoC*. 69
5.7 Analysis of the memory bandwidth exploitation improvements compared to PREM-like schemes. The tasks under test run on *ARM Cortex R5F* of the *Zynq UltraScale+ MPSoC*. 70
5.8 Analysis of the memory bandwidth exploitation improvements compared to PREM-like schemes. The tasks under test run on *ARM Cortex A72* of the *Versal ACAP*. 71
5.9 Analysis of the memory bandwidth exploitation improvements compared to PREM-like schemes. The tasks under test run on *ARM Cortex R5F* of the *Versal ACAP*. 72
5.10 ACT based on AXI Performance Monitor (APM) used to compare with our methodology. 73
5.11 SW and HW Throttling Costs (in *us*) to partition a DMA transfer in multiple smaller transfers. 74
5.12 Comparison of our tightly-coupled solution with loosely-coupled SW mechanism, to follow a temporized bandwidth profile. 77
5.13 Co-scheduling of LOW sensitivity tasks when running 1 and 2 concurrent ACTs. 81
5.14 Co-scheduling of HIGH sensitivity tasks when running 1 and 2 concurrent ACTs. 82

6.1 The F1/10 vehicle prototype. 87
6.2 Scheme and timing breakdown of the reference Particle Filter executed on a Xilinx ZCU102 board. 89
6.3 Proposed FPGA-overlay architecture. 89
6.4 System organization and Ray Marching Engine internal architecture, acting as co-processor for the PS. 90
6.5 Example of a Neural Network for objects detection, a typical application for autonomous systems perception. 93
6.6 Comparisons in terms of throughput (FPS) and accuracy (mAP), considering different pruning levels. 96

6.7 Benefits in terms of throughput (FPS), when multiple parallel streams are used. 97
6.8 Breakdown of the different phases a typical neural network is composed. As highlighted memory transfers play a big role on the total latency. 98

List of Tables

5.1 Programmable area needed to implement our RBR design. 64
5.2 Overheads of the software monitoring routine presented in Listing 5.5 76
5.3 Real-world benchmarks 79
5.4 PolyBench kernels grouped on two different taskset, based on interference sensitivity. 80
5.5 Synthetic memory-intensive workloads 83

6.1 Latency breakdown expressed in milliseconds of Particle Filter Algorithm, varying the number of particles (p). 91
6.2 Resource Usage of different designs 91
6.3 Floating Point versions of YOLO, varying datasets on GPGPU 94
6.4 Quantized versions of YOLO, varying datasets on FPGA 94

List of Abbreviations

CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPGPU	General-Purpose computing on Graphics Processing Unit
FPGA	Field Programmable Gate Array
ASIC	Application Specific Integrated Circuit
SoC	System-on-Chip
NoC	Network-on-Chip
HeSoC	Heterogeneous System-on-Chip
MPSoC	Multiprocessor System-on-Chip
ACAP	Adaptive Compute Acceleration Platform
ACT	Accelerator Cluster Template
RBR	Runtime Bandwidth Regulator
CLI	Cluster Load Intensity
PREM	Predictable Execution Model
CMRI	Controlled Memory Request Injection
MC	Memory Controller
LLC	Last Level Cache
AMBA	Advanced Microcontroller Bus Architecture
AXI	Advanced eXtensible Interface
APM	AXI Performance Monitor
DMA	Direct memory access
DRAM	Dynamic Random Access Memory
BRAM	Blocked Random Access Memory
IP	Intellectual Property
HLS	High Level Synthesis
PL	Programmable Logic
PS	Processing System
PSU	Processing System Unit
APU	Application Processing Unit

RPU	Real-Time Processing Unit
QoS	Quality of Service
COTS	Commercial-off-The-Shelf
FSM	Finite State Machine
RTOS	Real-Time Operating System
TCMT	Tightly-Coupled Monitoring and Throttling
LCMT	Loosely-Coupled Monitoring and Throttling
CNN	Convolutional Neural Network
AI	Artificial Intelligence

Chapter 1

Introduction

Cyber-Physical Systems (CPS) are one of the most significant advances in the development of computer engineering, electronics and communication technologies. With the CPS acronym we mean an integrations of computation and the surrounding environment. Embedded computers are small digital devices that are traditionally adopted as central brains for CPS, where their purpose is to monitor and manage few special functions, like sensing and acting based on input stimuli. Unlike classic personal computers, that are *general purpose*, embedded systems are small, lightweight and power constrained devices, that are programmed to perform few control tasks, like measurement and monitoring functions (*special purpose*). Embedded devices can be categorized into three different main areas: *Programmable Logic Controllers (PLCs)*, *Microcontrollers (MCUs)* and *Systems-on-Chips (SoCs)*. The PLCs, thanks to their rich connectivity to the outside, are systems that are designed to work in industrial environments, typically as actuators or monitoring systems (eg. to monitor and actuate hydraulic cylinders, relays or electric motors).

MCUs are another category of embedded systems, which unlike traditional CPUs are characterized by a lower working frequency, lower energy consumption (typically in the order of 1Watt, compared to the 30-110Watts of general purpose CPUs) and a rich connectivity to the external environment. MCUs are used in all those contexts where it is necessary to have digital systems for monitoring and actuation. A typical example involves MCUs as electric motor controllers [1]. A category of embedded systems that has experienced a significant growth in the last years is the high-performance *Systems-on-Chips (SoCs)* domain [48, 49, 47, 87, 78]. The definition of SoC derives from the fact that all or almost all the components of a system are integrated inside a single package, including: CPUs, GPUs, Decoders/Encoders, DSPs and many others. Unlike the previous categories, these systems are typically

[1] https://docs.px4.io/master/en/flight_controller/pixhawk4.html

employed not only for carrying out control and monitoring tasks, but they also have the computational capacity necessary for solving complex problems. A general organization of a CPS can be seen in Figure 1.1. Applications of CPS arguably have

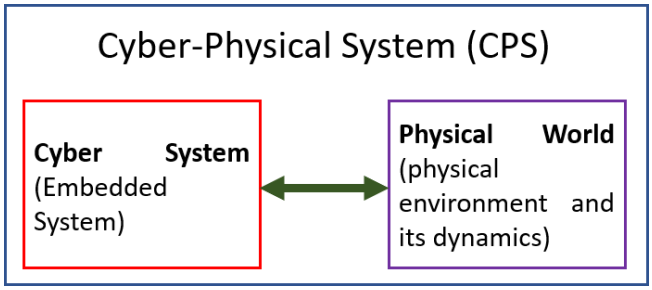


FIGURE 1.1: General Cyber-Physical System, composed of the physical world and an Embedded System used as its central compute environment

the potential to dwarf the 20-th century IT revolution and the potential of CPS to change every aspect of our life is enormous [32]. They include transportation systems and autonomous vehicles, distributed robotics (eg. robotic surgery, telepresence and telemedicine), defense systems, manufacturing and smart structures [41]. For example with the term *Industry 4.0*, we mean a paradigm, enabled by modern CPS, that encourages the usage of smart sensors, devices, and machines, to enable smart factories that continuously collect data pertaining to production [24, 56]. For human beings it is extremely difficult to guarantee the quality of every single product in particular at high production rates. The automotive sector could benefit considerably from the better integrated on-board intelligence enabled by CPS, which could improve safety and efficiency. Autonomous driving is a promising automotive field which has gain a lot of interest in the last few years. Self driving cars are an example of CPS, where, by sensing the environment with onboard cameras, LIDARs and other sensors are able to navigate without the human intervention. Self driving cars could have a huge impact of our lives, US government estimates suggest 94% of traffic collisions are caused by human errors [69]. Recent studies estimate that making 90% of cars on US roads self driving would save *25000* lives per year [36].

These types of systems need to get processed answers to input data following *"fixed timings"*, within pre-determined time windows called *deadlines* and after that the output is no more useful. For example, the braking system of a car, if fails to be almost instantaneous on every situation, could compromise the safety of the entire system. To avoid malfunctions, the electric motor control system must respond with

precise timing. Industrial monitoring systems must take the necessary countermeasures in time, when a certain event occurs. It is important to note that this feature has nothing to do with the computational power of the system. Embedded systems have always been held to a higher reliability[2] and predictability standard than traditional server or desktop systems, that fail to provide temporally predictable outputs.

Deep Learning (DL) and *Artificial Intelligence (AI)* represent an enabling technology of modern and intelligent CPS, especially for autonomous systems. *Deep Neural Networks (DNNs)* have a tremendous effectiveness for tasks as object detection and categorization. For example, in traffic sign recognition, the accuracy of a dedicated CNN can reach up to 99% of accuracy, which is better than human beings [85]. DNNs are key modules of the perception and localization systems of automated vehicles of tomorrow. To sustain the inherent complexity, hence, computational demand, of DNNs on size and power-constrained embedded platforms, typically system designers employ highly optimized *Convolutional Neural Networks (CNNs)*, where convolutional layers extract significant features from images. Nicely, the main operations that CNNs carry are sums and multiplications, more specifically defined as *Multiply Accumulate Operations (MACs)*, which are highly scalable with the number of neurone, and data-parallelizable, hence perfectly suitable for dedicated accelerators. To cope with the increasing computational demand of modern CPS, hardware vendors like NVIDIA, Xilinx or Intel are moving toward a *heterogeneous design*, where general purpose processors are coupled to application specific accelerators.

1.1 The heterogeneous systems revolution

The technological evolution antecedent to high-performance heterogeneous systems is long and complex. It was 1971, a historic date that marks the birth of the first microprocessor in the computer history: the Intel 4004, a processor characterized by a clock frequency of 750KHz, about four orders of magnitude lower than the frequency of today's processors. Since then, a myriad of optimizations and new inventions have led to physical limits impossible to overcome by single-core processors, the **Power Wall**, **ILP Wall** and **Memory Wall** [52]. 2005 marks the release of the first multi-core processor and the world has never been as before, since it was the first practical evidence that Moore's law was still valid. However, even the era of multi-core has inevitably reached the dawn, in fact with the increasing complexity of the

[2] for example the *Automotive Safety Integrity Level (ASIL)* is a risk classification scheme defined by the ISO 26262, that defines the functional safety for road vehicles.

multi-core systems, another physical limit has been reached on the maximum use of silicon area capable of switching to the maximum frequency, without compromising heat dissipation. This limit is known as **Utilization Wall** [72]. Is defined **Dark Silicon** the portion of the chip that cannot be exploited at their maximum frequency due to physical constraints. The heterogeneous systems *"are the Specialized Horsemen of the Dark Silicon Apocalypse [72]"* and represent one of the possible answers to the question *"is Dark Silicon useful?"* They are characterized by a limited number of general purpose processors (typically 2, 4 or 8), coupled by application-specific engines, capable of solving specific problems in an energy-efficient way. ARM big.LITTLE[3], is the most popular example of heterogeneous embedded system of today. The heterogeneous design, although born mainly for the embedded domain, by few years has also been affecting the desktop and server domain. NVIDIA with the RTX series[4] has recently integrated various types of GPGPU cores within its chips (eg. regular and tensor cores). From a desktop processor perspective, Intel has released *Alder Lake*[5] at the end of 2021, the first desktop processor based on a heterogeneous architecture. Heterogeneous systems, from one side provide sufficient computing capability to satisfy the needs of modern software applications like neural network inference and training. The adoption of a heterogeneous design for the embedded domain, from the other side, allows to reach an improved energy-efficiency compared to multi-core systems.

1.2 The other side of the coin

Commercial-off-the-Shelf (COTS) heterogeneous embedded systems are an appealing choice for application companies to implement or prototype CPS. With the COTS term we refer to hardware platforms developed by a supplier for multiple customers, whose design and configuration is controlled by the supplier's or an industry specification [73]. COTS heterogeneous systems may include I/O devices, microprocessors, FPGAs, GPUs or other digital components. This represents one of the main reasons that led to the boom of heterogeneous embedded systems, as this market strategy, which does not require to commission custom-made components by application and system developers, allows the reduction of chip production costs. This lead to a lot of benefits including a more rapid introduction of the product on

[3] https://www.arm.com/why-arm/technologies/big-little

[4] https://www.nvidia.com/en-gb/geforce/rtx/

[5] https://www.intel.com/content/www/us/en/products/docs/processors/core/12th-gen-processors.html

the market, a reduced development time, a lower life-cycle costs by taking advantage of up-to-date commercial industrial base and many others.Chip manufacturers like NVIDIA or Xilinx, adopt a *shared resources* architectural template, where different hardware IPs contend to access resources (see chapter 2). These systems are typically interconnected in shared buses, can access shared local memories (caches and scratchpads), and also share the Memory Controller (MC) and the main memory (DRAM) as the main source of communication between heterogeneous processing engines. This architectural organization is adequate only when the load (i.e. the usage in a given time interval) does not exceed a certain critical threshold, beyond which the performance of the system undergoes a performance degradation which is difficult or impossible to predict, caused by contention phenomena on shared resources. The most critical of these shared resources is the main memory (DRAM), whose maximum bandwidth is typically far below the rate of requests that the processing elements of a system are able to generate. This represents a problem that complicates the adoption of these systems for applications where runtime guarantees must be provided, and a deadline failure could compromise the safety of the entire system. Figure 1.2 shows the memory requests profile (expressed in Mega

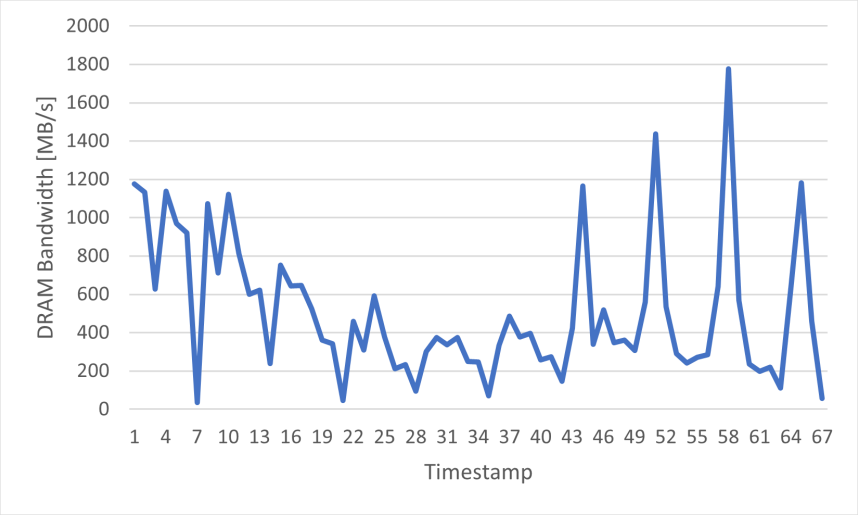


FIGURE 1.2: Main memory bandwidth requests over time required by the Xilinx DPU accelerator. The neural network performs a *pose detection* task on Zynq UltraScale+ device.

Bytes per second) of a neural network for *pose estimation*, a problem that is typically found in the context of computer vision and robotics, running on the Zynq UltraScale+ platform, a heterogeneous FPGA-based system. This motivational example shows that neural networks, on these types of systems behave as memory intensive applications. On heterogeneous systems, where different types of computing

elements coexist within the same chip, this could lead to contention problems. This justifies the study of regulation and interference mitigation mechanisms for this type of applications.

1.3 Contributions of this research

Several research groups studied the problem of resource contention on multicore CPUs [54, 75] and GPGPU-based heterogeneous embedded devices [80, 12, 15]. Various techniques and approaches have been proposed in the literature, some of which are based on more or less pessimistic resource access protocols [55, 14]. From the point of view of FPGA-based heterogeneous systems, some works go in the direction of mitigating interference on shared interconnects [60, 62]. Other approaches that focus on regulating main memory bandwidth are however based on platform-specific mechanisms [86, 67]. The main contributions of this thesis are:

1. Characterization of memory interference on high-end embedded devices. In this investigation, the main memory interference that the processing elements of a heterogeneous system are affected has been studied and used to motivate the integration of mitigation mechanisms on these types of systems. The interference analysis involves two different levels of the memory hierarchy: starting from the main memory (DRAM), which is the last level of the hierarchy. Finally the experiments involve also the shared *Last Level Cache (LLC)* interference. The analyses were conducted on two different types of state-of-the-art embedded platforms: i) the Xilinx Zynq UltraScale+ and the Xilinx Versal ACAP, respectively two FPGA-based platforms. ii) the NVIDIA TX2, a heterogeneous system that combines a multi-cores processor and a GPU. The most relevant findings are presented in chapter 4. The memory interference problem is less explored on FPGA-based heterogeneous systems and this is one of the first study that presents an extensive experimental campaign on this type of systems;

2. Driven by the experimental analysis proposed in chapter 4, a novel methodology to mitigate such a problem is presented. chapter 5 presents the *Runtime Bandwidth Regulator (RBR)*, a *Tightly-Coupled* system to monitor & regulate the memory bandwidth exploited by FPGA accelerators. The *RBR* is based on *Controlled Memory Request Injection (CMRI)* [16], a technique that make use of throttling to regulate the memory bandwidth. This work fills a gap in the literature,

where: i) most of the available techniques inefficiently exploit the available memory bandwidth; ii) other regulation mechanisms, are too *Loosely-Coupled* and do not allow a fine regulation of the available bandwidth, limiting its usage by the accelerators of the system; iii) often, fine-grained regulation mechanisms are platform-specific and address the problem only for some specific platforms. These aspects are addressed in chapter 5;

3. Finally, chapter 6 presents two applications in the embedded domain, where the techniques proposed in chapters 4 and 5 can be effectively applied. The first application considered in this thesis involves the design of an FPGA accelerator used for the localization of a robot within an environment. The design of this accelerator has been built following the architectural template considered in chapter 4. The second application presented is an implementation and evaluation effort to deploy *Convolutional Neural Networks (CNNs)* as perception for autonomous systems. Also in this second use-case the mechanisms and knowledge presented in the previous chapters can be effectively exploited.

Chapter 2

Overview of Embedded Heterogeneous Systems

2.1 General Template

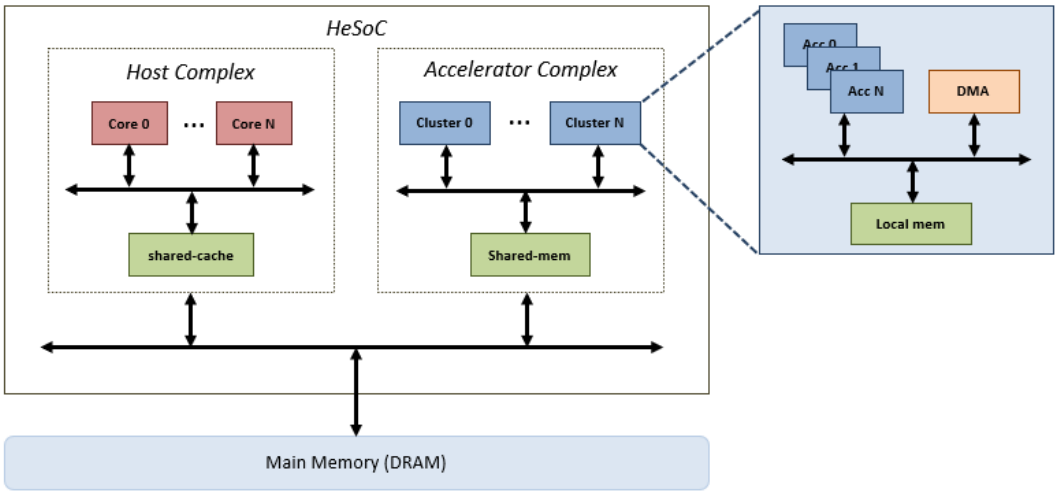


FIGURE 2.1: General architectural template for heterogeneous SoCs, comprising both host cores and accelerators.

Figure 2.1 shows the general block diagram used on modern Heterogeneous Systems on Chips (HeSoCs) considered in this thesis. As shown, different processing elements (PEs) are coupled inside the same chip, each of which is used for different types of workloads. The *Host Complex* contains a certain number of processor cores typically with a *reduced instruction set (RISC ISA)*, where each core integrates a certain number of dedicated registers and a private L1-cache to the core itself. The cores are interconnected to a shared cache (typically second level), through a shared bus. The main bus is in turn interconnected to the main memory, which represents the last level of the memory hierarchy, through the *Memory Controller (MC)*, a digital circuit that orchestrates the flow of data going to and from the main memory. Host cores

typically use *Multiple-Instructions-Multiple-Data (MIMD)* parallelism and perform control tasks. However, modern processors also support *Single-Instruction-Multiple-Data (SIMD)* parallelism by means of dedicated instructions. Next to the host subsystem the *Accelerator Complex* can be found, where one or more data-cruncher engines are clustered each other. Each cluster is organized as in the blue box shown in Figure 2.1, where one or more accelerators used for carrying out computation, a DMA and a local memory are integrated inside each cluster. Accelerators are digital circuits dedicated to solve specific problems in an energy-efficient manner. Heterogeneous systems integrate this type of engines as the execution of compute-intensive problems (like data-parallel) on the host complex would be too inefficient. The Direct Memory Access (DMA) is a fundamental component that is typically integrated inside the accelerator clusters. DMAs are dedicated circuits that can be exploited to perform memory transfers, allowing host processors to perform useful computation in parallel with memory transactions. The Local-memory, typically order of magnitude faster than shared memory, is directly accessible by accelerator clusters and DMA and can be composed of several memory-banks. Local memories are the first memory level closer to accelerators and are typically organized as a *scratchpad*, i.e. without the control circuitry typical of cache memories. Following this organization, the overhead introduced by the coherency protocols can be avoided. Furthermore, the reduced control circuitry allows to decrease the power consumption and the silicon area needed to implement the chip. The clusters are also interconnected to the DRAM using a certain number of MC ports, typically exploiting a number of ports reserved for accelerators. For example the Zynq UltraScale+ MPSoC has three different DRAM ports reserved for the Programmable Logic.

The systems targeted in this work are organized according to this heterogeneous architectural design, where a *host complex*, composed of general-purpose CPUs, is coupled to efficient accelerator subsystems. These components are interconnected via a hierarchy of buses and crossbars (or a network-on-chip) to a shared DRAM controller. As a representative embodiment of such a general template two *commercial-of-the-shelf (COTS)* platforms from Xilinx have been chosen: the *Zynq UltraScale+ MPSoC* [87] and the *Versal ACAP* [78] and two platforms form NVIDIA: the *NVIDIA TX2* [49] and the *NVIDIA Xavier AGX* [47]. All the systems considered adhere to the architectural design described above. In this section the main architectural features of the reference platforms are introduced.

2.2 NVIDIA Tegra X2

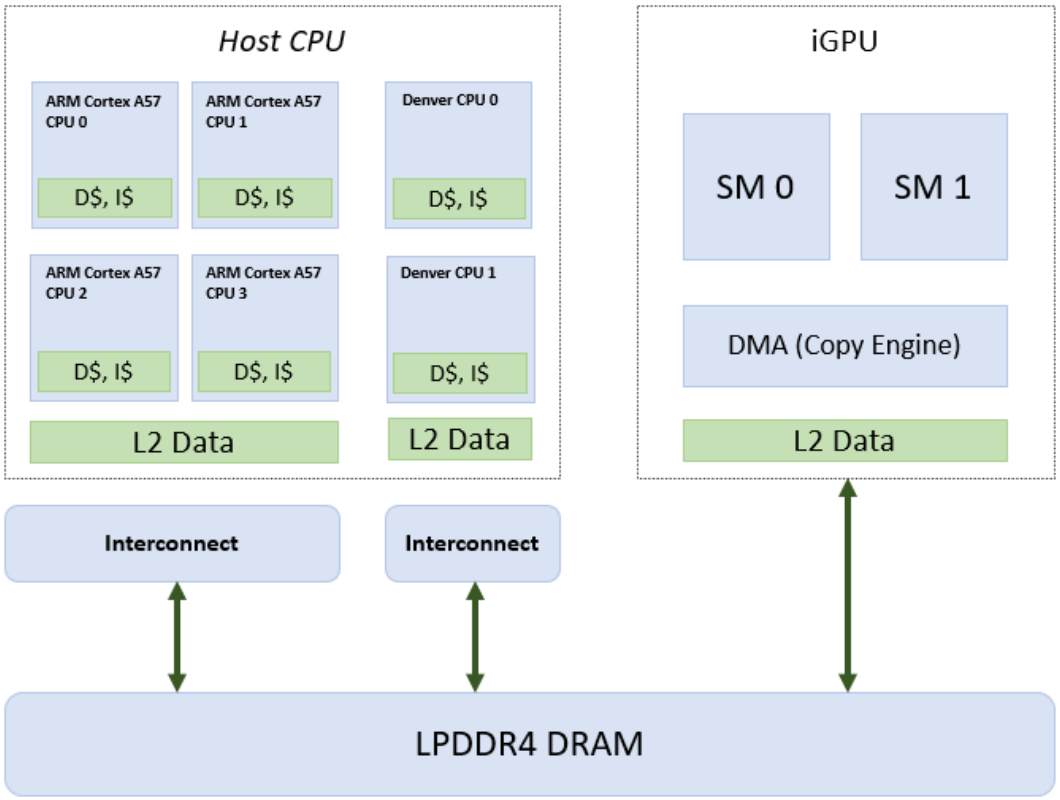


FIGURE 2.2: Block diagram of the NVIDIA Tegra X2.

The NVIDIA Tegra X2 platform is a commercial SoC developed by NVIDIA and released in 2017. From the point of view of the CPU it is composed of two different islands of cores: an ARMv8 Cortex A57 quad-cores processor and an ARMv8-compliant processor called Denver, developed by NVIDIA itself. The NVIDIA TX2 platform follows a design called Big.SUPER, in which the Denver processor, characterized by a more aggressive Out-of-Order instruction issue than the ARM A57, is used to manage more complex computational loads. Despite the difference between the two islands of cores, the working frequency, set at 2GHz, is the same in both cases. Each of the six cores present inside the SoC integrates an L1 data cache of 32KB and an L1 instruction cache of 48KB. As for the second level last level cache (L2-LLC) there are two separate banks of 2MB size for each island. The accelerator integrated inside this SoC is an iGPU part of the Pascal micro-architecture, in which 256 CUDA cores are divided into two different Streaming-Multiprocessors (SMs). The maximum clock frequency that can be reached by the iGPU cores is 1.3GHz. The host processors and the iGPU are interconnected to a shared central memory

LPDDR4 DRAM of 8GB 128-bit width at a maximum frequency of 1866MHz, capable of providing a maximum bandwidth of 59.7GB/s. Figure 2.2 shows the block diagram of the platform described above.

2.3 NVIDIA Xavier AGX

Figure 2.3 shows the block diagram of the NVIDIA AGX Xavier platform, a platform released by NVIDIA in 2018 as a direct successor to the previous NVIDIA Tegra X2. From the point of view of the CPU, the NVIDIA AGX Xavier, unlike the previous generation, is composed of a single island that integrates eight identical ARMv8.2-compliant processors, called *Carmel* and produced by NVIDIA as successors of the *Denver* processors. The maximum working frequency of the Carmel cores is 2265MHz and they integrate optimized OoO execution policies. From the memory hierarchy point of view, Carmel processors integrate 4 identical islands in which there are two Carmels that share a 2MB L2 cache. Each core also has a 64KB-wide private L1 cache internally. Unlike the previous generation, the cache hierarchy of the Xavier AGX SoC is composed of three cache levels, the last of which (L3-cache) is shared among all the cores of the CPU Complex. The iGPU is a *Volta* micro-architecture, characterized by a maximum frequency of 1377MHz, whose respective 512 CUDA cores (double compared to the previous generation) are divided into 8 SMs, each of which groups 64. It is also important to remember the presence of Tensor Cores, particular CUDA cores inside the GPU optimized for carrying out operations of Deep Learning and neural networks. Xavier AGX integrates 16GB of LPDDR4 with a bus width of 256-bit. The maximum frequency that the memory subsystem can reach is 2133MHz, which implies a maximum memory bandwidth of 137GB/s. The NVIDIA Xavier platform also integrates two different types of ASIC accelerators. The first called *Video Processing Unit (VPU)* is an accelerator dedicated to the execution of video processing algorithms, while the second type, called *Deep Learning Accelerator (DLA)* can be used to accelerate neural networks inference.

2.4 Xilinx Zynq UltraScale+

The *Zynq UltraScale+ (ZUS+)* is a family of Multi-Processor SoCs (MPSoCs) that comprises several types of chips each with unique characteristics. The ZU9EG is the flagship MPSoC of the ZUS+ family which is embedded at the heart of the *Xilinx*

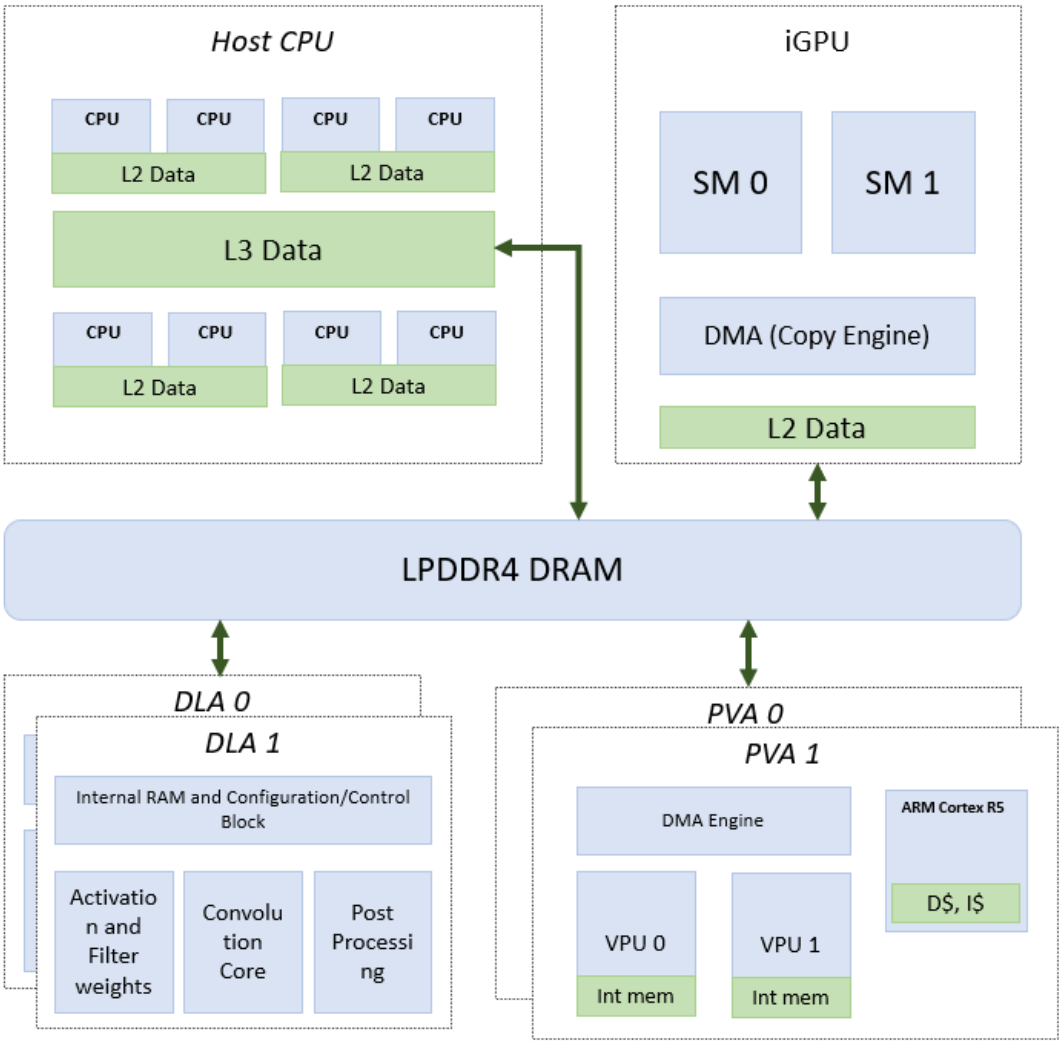


FIGURE 2.3: Block diagram of the NVIDIA Xavier AGX.

ZCU102 development platform[1]. As shown in Figure 2.4 this chip is composed of a heterogeneous *host* processor *(PSU)* for general-purpose and real-time computing alongside a *Programmable Logic (PL) block* (FPGA) for the acceleration of compute-intensive kernels. The PSU is composed of two processor islands, named *APU* and *RPU*. The *APU* consists of a quad-core *ARM Cortex A53* processor implementing the *ARMv8-A* micro-architecture, and constitutes the target choice for general-purpose computing. As shown in Figure 2.4, each core features 32KB private L1 cache for instructions and data. The four cores share a 1MB L2 cache with 64B-wide lines, and can access the memory controller via a *Cache Coherent Interconnect*. The maximum operating frequency of this processor is 1.3GHz.

[1] https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf

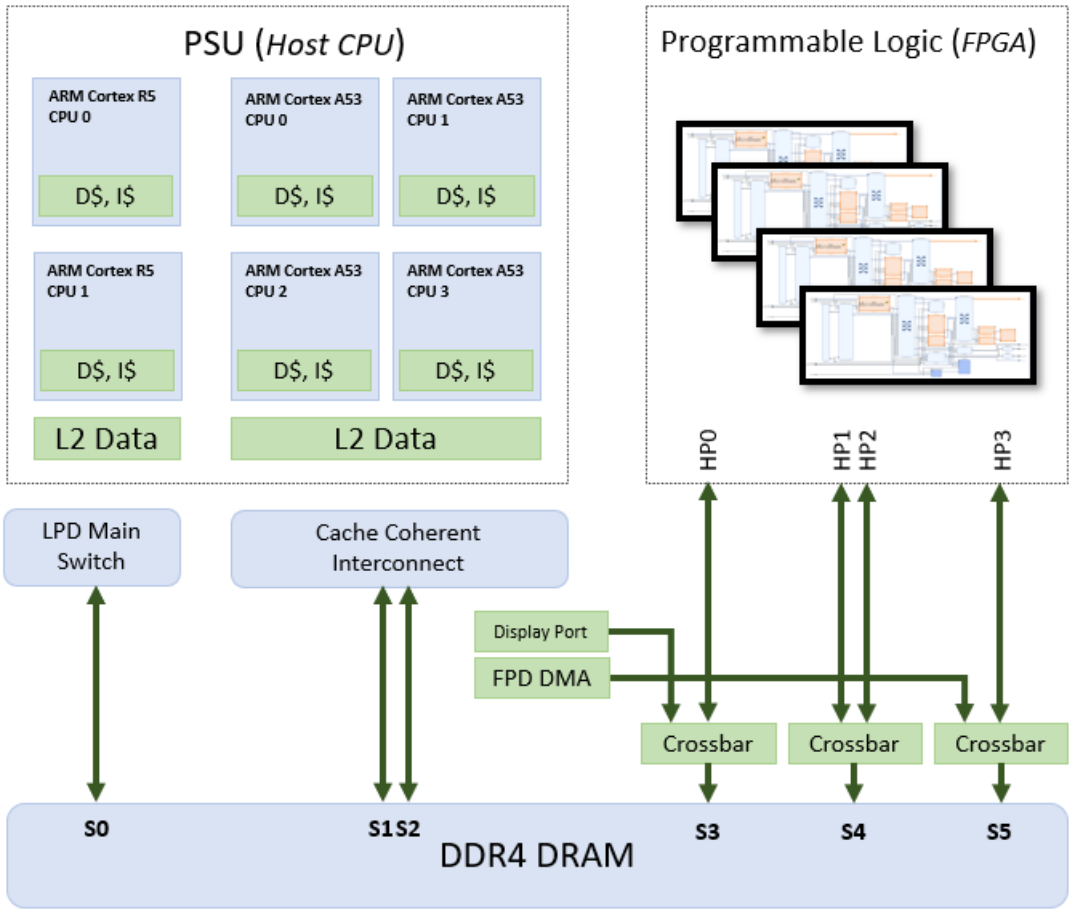


FIGURE 2.4: Block diagram of the ZU9EG MPSoC, the chip within the Xilinx ZCU102 evaluation platform.

The *RPU* consists of a dual-core *ARM Cortex R5* dual-core, which is used to deploy control tasks with real-time requirements. Each R5F core has a private data cache 64KB-wide, characterized by a single-line size of 32B. This processor also features a *Tightly Coupled Memory (TCM)* for workloads that must remain resident. The RPU can be used in *Lockstep* to implement a so called *Dual Modular Redundancy*, where the two cores executes the same instruction at the same time. To test the maximum interference the RPU was set in *Split Mode*, a working mode where each core is independent. The working frequency of Cortex R5F cluster is 500MHz, for this reason this processor is typically not capable of sustaining high-performance workloads, then little stress on the memory hierarchy is expected. Inside the PSU there is also an ARM Mali-400 MP2, a GPU that unlike the NVIDIA-based, can be used for graphics only.

The PL block is interfaced with the outer world via different types of AXI4 ports, some of which allow for coherent transactions with the PSU through the CCI, while some others leverage a dedicated path to the DRAM controller. In this work, the

latter was considered, called the *AXI High-Performance Ports (HP)*, which can sustain the maximum bandwidth to the DRAM [44]. Each of the HP ports has a width of *128 bits* and can reach a maximum nominal frequency of *300MHz*. The paths originating from the four AXI HP ports are multiplexed via a hierarchical crossbar into three ports into the memory controller. Specifically, each crossbar performs arbitration between two different entities: *i)* Display Port and HP0; *ii)* HP1 and HP2; *iii)* FPD DMA and HP3. FPD DMA and Display port are not used in the work presented in this thesis. The *Speed Bin* used to configure the DRAM is *DDR4 2133P*, with a bus width of 64B. The working frequency of the memory is *1066MHz*. Thus the maximum bandwidth is equal to: $1066 \times 2 \times 64/8 = 17GB/s$.

2.5 Xilinx Versal ACAP

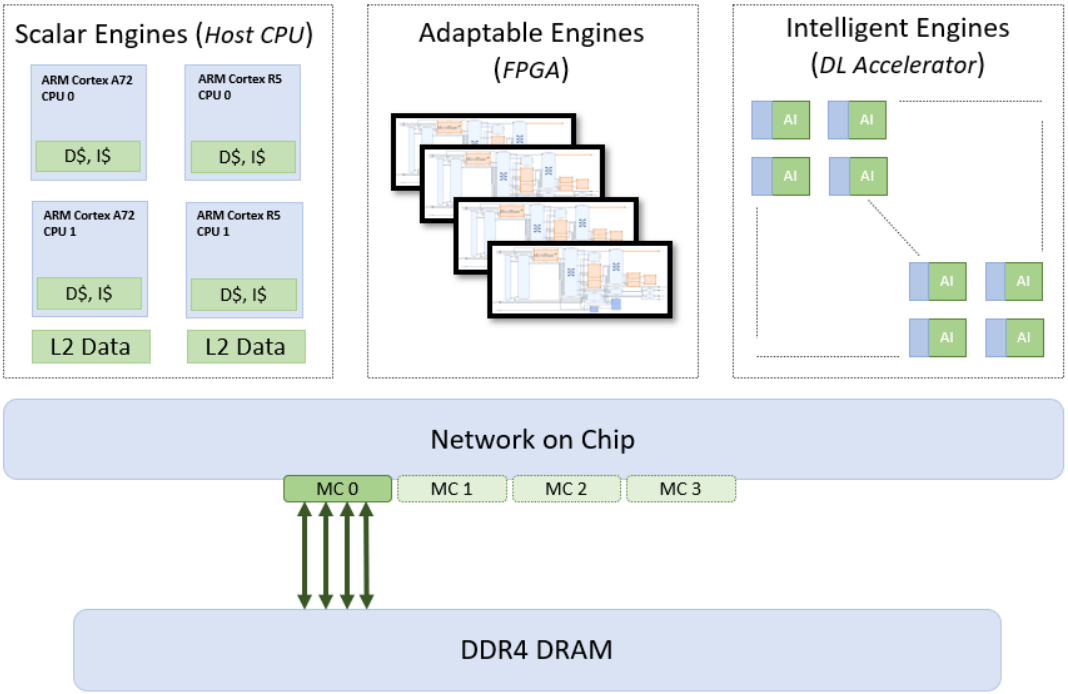


FIGURE 2.5: Block diagram of the VC1902 ACAP, the chip within the Xilinx VCK190 evaluation platform.

The VC1902 ACAP, at the heart of the *Xilinx VCK190* development platform[2], represents the new generation of HeSoCs belonging to *Versal ACAP* family by Xilinx. The concept of heterogeneity is further developed in this SoC family by combining different types of computational units with a more traditional FPGA fabric. As highlighted in Figure 2.5 this chip is composed of three main computational units:

[2] https://www.xilinx.com/support/documentation/boards_and_kits/vck190/ug1366-vck190-eval-bd.pdf

Scalar Engines, *Adaptable Engines* and *Intelligent Engines*. The *Scalar Engines* are themselves a heterogeneous system consisting of an *ARM Cortex A72* dual-core processor for general-purpose computing and an *ARM Cortex R5F* dual-core processor for real-time workloads. The ARM A72 cluster, is characterized by a 7nm technology process, which guarantees an increase in performance/watt of about 2× compared to the previous generation. The Cortex A72 implements the ARMv8-A microarchitecture, which for the VC1902 runs at up to 1.35GHz. Each cores has a private L1 cache composed of 48KB for instructions and 32KB for data. Cores are clustered in two-unit blocks sharing 1MB L2 cache featuring 64B line width. The RPU is the same as the ZU9EG MPSoC. The *Intelligent Engines* are composed of a fabric of VLIW cores organized as a systolic array and aimed at increasing the performance of Deep Learning workloads. Globally, this block is also called *AI Engine (AIE)*, and is organized according to a two-dimensional topology of *AI Engine Tiles*. Each tile includes a module containing AXI4-stream and AXI4 memory mapped interfaces, a memory module and an AI Engine core. The AIE subsystem is connected to main memory by means the FPGA. On the presented setup the AIE and main memory was interfaced through the *Deep Learning Processing Unit (DPU)* IP [3] which also implements some layers not supported yet by the AIE. Finally, the FPGA fabric in this platform is called *Adaptable Engines*, and constitutes the main focus of this work.

The three computational units can access DRAM through a global *Network-on-Chip (NoC)*. The configuration of the NoC can be done by the designer, with the *AXI NoC* IP provided by Xilinx. Through the AXI NoC it is possible to connect the Versal engines to two different memory subsystems within the platform, namely the *LPDDR4* and the high-performance *DDR4 DRAM*, which has been used in the experiments shown in the following chapters. As shown in Figure 2.5 the AXI NoC component supports up to four *Memory Controllers (MC)*, however only one MC has been enabled in this evaluation as the VC1902 embeds only one MC. Unlike on Zynq UltraScale+ which has only four HP ports, on Versal ACAP it is possible to enable a greater number of ports to interconnect the FPGA to the NoC. Also in this case the width of the AXI4 ports is 128 bit and their maximum working frequency is 300MHz. The reference *Speed Bin* that adopted to configure the DDR4 DRAM is *DDR4-3200AA*, the bus width is *64 bit* and the working frequency is fixed to *1600MHz*. These parameters imply a maximum memory bandwidth equal to $1600 \times 2 \times 64/8 = 25.6GB/s$.

[3] https://www.xilinx.com/products/intellectual-property/dpu.html

Chapter 3

Related Work

3.1 Main topics

As previously outlined in the introductory section, the main topics covered in this doctorate thesis are: i) the memory interference characterization on heterogeneous systems; ii) techniques to mitigate the memory interference problem on FPGA-based HeSoCs. For this reason the related work chapter follows this organization and it is subdivided into two different parts. For each part are reported the main advances that have been achieved in recent years, from the point of view of the characterization of new generation heterogeneous platforms and interference on shared memories. Subsequently an analysis of the main works in the literature is proposed as regards the mitigation of the problem of interference of shared resources, with particular reference to the main memory.

3.2 Memory interference and platform characterization

A solid and accurate methodology to mitigate memory interference should start from an accurate platform and interference characterization, this is especially true for heterogeneous systems, where different processing elements coexist on the same chip. Different processors and accelerators differ significantly in terms of main memory access patterns and memory requirements. The problem of memory interference has been extensively studied in the context of multi-core CPU and GP-GPU HeSoCs, but is less investigated on FPGA-based heterogeneous systems, which is part of this dissertation.

3.2.1 CPU and GPGPU platforms

R. Cavicchioli et al. [15] qualitatively analyzed and characterized the conflicts due to parallel accesses to main memory by both CPU cores and GPGPUs. The authors considered two embedded platforms: the NVIDIA TK1 and NVIDIA TX1 and a desktop environment composed of an Intel i7-6700 with an integrate GPU (iGPU). They showed that CPU activities can severely affect GPU task execution time.

N. Capodieci et al. [12] extend their previous work [15], by studying the interference problem on different generations of NVIDIA SoCs, analyzing how these architectures have evolved from the point of view of memory contention. Their analysis shows that memory contention remains the most important issue on both a performance and predictability perspective. Hao Wen and Wei Zhang conducted similar explorations [80], also focusing on CPU-GPU Systems. Their work shows that the CPU workload could be unfavorably prioritized over the GPU and could lead to significant slowdowns. These papers are specific to GPU-based systems only and do not explore solutions for interference mitigation.

D. Tarapore et al. [71] proposed a Linux kernel module that, based on the hardware support of the RAMINDEX IP, can snapshot the cache content of ARM-based systems. The authors leveraged their monitoring infrastructure to characterize cache interference on NVIDIA Tegra X1 SoC and to build a cache-interference predictor.

C. Courtaud et al. [18] propose an experimental characterization of applications memory consumption to determine their sensitivity to memory interference, by adopting a configurable micro-benchmark able to reproduce a wide range of memory consumptions and access patterns. The outcome of their research is that a purely quantitative characterization lacks in precision and qualitative information should also be integrated. Their work focuses on multi-cores CPUs only and do not take into considerations heterogeneous systems nor accelerators.

3.2.2 FPGA-based platforms

Very few papers have studied the effects of memory interference in the context of FPGA-based HeSoCs. A. Bansal et al. [5] characterize the memory subsystem of the ZU9EG MPSoC, through micro-benchmarks. Their research highlights that on-chip *Block RAMs* are less affected by interference than other memories. They also propose a hardware/software infrastructure capable of guaranteeing isolation. K. Manev et al. propose a characterization of the memory subsystem of two SoCs belonging to the *Zynq UltraScale+* family: ZU9EG and ZU3EG [38]. Their experiments involve

memory-mapped accelerators interconnected in DRAM through the *AXI HP* ports. The work presented is an exhaustive characterization of the throughput obtained through the AXI HP ports according to different parameter configurations. However unlike the exploration proposed in chapter 4, the issue of memory interference between Host CPUs and FPGA is not addressed.

M. Mattheeuws et al. [43] conduct a similar analysis on the Xilinx ZU9EG SoC, evaluating the performance slowdown that a task running on the ARM cluster is affected when traffic generators instantiated on the FPGA contend for the DRAM bandwidth. While the aims are similar to what is proposed in chapter 4, here the analysis is overall less comprehensive, carried out on a single SoC and neglectful of solutions to mitigate the effects of interference.

Focusing on cache memory interference, D. Hoornaert et al. [27] discovered a new source of interference they called CPU-brainfreeze. Applications may suffer up 10× of execution time increase or indefinitely stalled in extreme cases.

The rest of this chapter proposes an overview of state-of-the-art hardware/software techniques to mitigate the memory interference problem.

3.3 Approaches to mitigate interference

Several methodologies have been proposed in the literature, they can be categorized into: i) *software schemes* for accessing shared resources. This category includes policies and algorithms for controlled accesses to shared resources; ii) *hardware extensions* including COTS hardware IPs integrated within the reference platforms or ad-hoc hardware extensions to mitigate the memory interference problem.

3.3.1 Software approaches

Concerning the study of techniques to mitigate memory at the software level, both the Certification Authorities [14] and a number research groups [55, 46, 2, 40, 22] have outlined mutually exclusive access to the DRAM as the practical solution to address the adverse effects of memory interference in shared-memory SoCs. These approaches can be categorized as *Phased Execution Models*, where the main idea is to refactor the tasks' code to create and equivalent task, that is decomposed into memory and compute phases. Each tasks starts by executing a memory phase, where the data are copied into local memories. Subsequent compute-phases work on local

data. Typically these types of task models are used in conjunction with scheduling policies that guarantee mutual exclusion on the shared memory accesses.

R. Pellizzoni et al. [55] defined the *Predictable Execution Model (PREM)*. Following a Phased Execution Model a task is decomposed into *compatible* and *predictable* intervals. Compatible intervals, reserved for code portions that the compiler can not determine the regions of accessed memory, from a scheduling point of view are treated as memory phases. The experiments carried out by the authors confirms that their model is effective at mitigating memory interference and to make the tasks' WCET more predictable.

J. Nowtosch et al. [45] introduced an interference-aware WCET analysis, where an offline static scheduling is assumed. The WCET calculation is decomposed on two different parts: the task execution time assuming a privileged access to share resources and an upper bound to the delay caused by interference. In their setup the Operating System monitors the resource usage and throttles the tasks when their budget is reached. The same authors extended their previous work [45], by implementing a budget re-claiming mechanism [46].

A. Alhammad et al. [2] proposed an offline static scheduler based on PREM, to avoid concurrent execution of memory phases to avoid interference on the shared memory. The authors extends the work of Pellizzoni [55], orinally proposed to mitigate interference between cores and I/O devices. In the work proposed by Alhammad the focus is to mitigate interference on multi-cores systems, under the assumption that each core has a private LLC or a partitioned shared cache. In later 2014 Alhammad proposed an extension of the original PREM model called global PREM (gPREM) [1]. The effort is to ensure that a single memory phase is executed system-wide.

B. Forsberg et al. [22] proved that the PREM model can also be effectively extended also on heterogeneous systems. The authors proposed the first PREM compiler for GPGPU-based HeSoCs, a compiler that transforms GPU kernels into PREM-compliant code. Their experimental evaluations showed that near-zero WCET variability has been achieved.

In general these approaches are typically too pessimistic, as their *one-at-a-time* execution model causes severe under-utilization of the system DRAM bandwidth [2, 70, 40]. Attempts to overcome such pessimism have been made by G. Yao et al [81], who propose to allow multiple tasks to access DRAM at the same time to increase the memory bandwidth utilization. The main drawback of this approach is

that task-based scheduling for memory accesses does not allow fine-grained control, failing to provide practical guarantees on the latency increase bounds.

A recently proposed technique aimed at solving the memory bandwidth under-utilization of mutually exclusive policies is *Controlled Memory Request Injection (CMRI)* [16], where, unlike the previous works, more than a single task at a time can access DRAM. The memory phases of such tasks are however implemented as controllable duty cycles, where memory requests are interspersed with idle cycles. Idleness insertion can happen both via compiler-level instrumentation of the program or via dynamic task throttling [83, 74]. In multicore systems, CMRI is effective at increasing bandwidth utilization without significantly increasing the latency of the main task.

3.3.2 Hardware extensions

While the above solutions rely only on software and scheduling strategies, HW and HW/SW solutions have been proposed to control and mitigate concurrency on the shared memory systems. They can be categorized into *custom hardware solutions*, that are hardware designs and extensions developed to mitigate the interference. In this category hardware arbiters or bandwidth regulators can be found. In the *platform-specific solutions* are grouped the latest works that focuses on mitigating memory interference by using platform-specific solutions developed by hardware vendors and integrated into their platforms.

FPGA-based approaches and custom HW solutions

F. Restuccia et al. focus their work on studying contention and fairness of shared buses. They propose an analysis of the *AXI Interconnect* IP of the Xilinx ecosystem where they showed that a critical issue that can arise when adopting the AXI protocol under round-robin arbitration and heterogeneous burst sizes. These conditions could lead to a completely unfair bandwidth distribution of the AXI masters connected to the shared interconnect that could imply in a bandwidth decrease. They propose the *AXI burst equalized (ABE)* to restore fairness [62].

M. Pagani et al. [51] proposed the *AXI Budgeting Unit (ABU)* a hardware based design to implement a bandwidth reservation mechanism on top of the AXI protocol, to regulate the bandwidth utilization of FPGA-based accelerators. The authors also proposed a theoretical task model to bound the response time analysis of the

hardware accelerators and an evaluation on a Xilinx Zynq-7020 SoC. The same research group integrated their regulator IPs [62, 51] into the *AXI HyperConnect* [61], an hypervisor-level component that allows interconnecting hardware accelerators into the same AXI bus while preserving bandwidth regulation and AXI fairness.

A recent work from F. Restuccia et al. [60] focuses on memory interference generated by instruction fetch and data read/write of the Xilinx DPU engine. Their idea is to partition the instruction and data memory, used by the DPU, to avoid self-interference. The memory partition has been done through a dedicated FPGA-based IP in between the engine and the memories. As alternative memory location, they leveraged the Zynq UltraScale+ *On-Chip Memory (OCM).*

The *Barcelona Supercomputing Center* proposed hardware IPs to monitor and mitigate interference [11, 13], and to stress corner cases with custom AXI traffic generators [64]. They mainly focused on soft processors, while the focus of this thesis is to regulate and mitigate memory interference on both host processors and accelerators.

F. Farshchi et al. [21] try to mitigate memory contention exploiting bandwidth regulations on host cores, by leveraging an FPGA-based component, in the middle between *host* complex and shared buses. The focus is always on multi-core systems, not on heterogeneous, FPGA-based accelerators. The focus of their work is to regulate the bandwidth of multiple cores, to improve the bandwidth utilization, by removing the overhead introduced by software regulation mechanisms. The authors evaluated their work using the FireSim, an FPGA-based full-system simulator [30].

Marangoz et al. [39] try to mitigate the main memory interference on GPGPU-based systems. They proposed a dynamic memory bandwidth reservation system that from one side mitigate can mitigate interference and from the other enhance strict partitioning mechanisms. The authors validate their system on simulators only.

D.Hoornaert addresses the problem of memory interference in real-time embedded systems based on programmable-logic [26, 63]. Their technique allows transaction-level control, redirecting the memory traffic generated by the host cores through the FPGA. The goal is to propose an infrastructure that allows to study custom policies and techniques for accessing the main memory. This mechanism allows to achieve an unprecedented degree of control at the level of memory transaction, by sacrificing the memory bandwidth, that due to HPM ports drops significantly.

Platform-specific solutions

Hardware vendors proposed HW IPs for controlling QoS and priority at different levels (i.e., interconnections like AXI CCI, DRAM memory controllers, etc.). Such knobs, as showed by Serrano et al. [67] can be not easy to use due to the many different, and in some cases obscure, configurations available. In the work proposed by Serrano, the QoS regulators of the Xilinx Zynq UltraScale+ MPSoC are evaluated and adopted to mitigate the memory interference while co-scheduling software and hardware tasks on a mixed-criticality system.

Sohal et al. [68] try to predict the temporal behaviour of workloads deployed on CPUs and accelerators. While the solution enables full bandwidth usage, an offline profiling procedure is required to calculate the scheduling and bandwidth allocation statically. In terms of memory bandwidth regulation, the authors leveraged a joint regulation using *MemGuard* [83] for the CPU cores and the *ARM QoS400* [4] to regulate the memory bandwidth of the accelerators.

Schwäricke et al. [66] tackle the problem of predictable data transfers between virtual machines, considering virtualized hardware resources on the Zynq UltraScale+ MPSoC. In their paper the authors present an architecture composed of several virtual machines based on *FreeRTOS*, where a particular virtual machine called *broker* is responsible for orchestrating communications. The interference mitigation infrastructure used by the authors is based on that proposed by Sohal [68], where MemGuard and ARM QoS400 are used to regulate the CPU and accelerator tasks respectively.

Zini et al. [86] studied the main memory contention generated by the I/O devices (like the Ethernet subsystem) as extensively used by many emerging applications, like autonomous vehicles. They investigates the capabilities of the ARM QoS400 regulator, to control bus memory bandwidth by means of a novel software infrastructure based on Linux. They analytically characterized and experimentally validated against theoretical bounds, the behaviour of QoS400 regulator, proving that they are effective in predictably controlling I/O-related memory contention.

Moreover, these mechanisms are typically closed solutions specific to particular hardware (i.e. *ARM MPAM* [53], or *Intel Resource Director Technology* [28]).

Chapter 4 builds upon these findings to study the problem of main memory interference in FPGA-based SoCs and LLC interference on NVIDIA Tegra platforms. Then a generic scheme is proposed to deploy CMRI on the FPGA, and thoroughly study the benefits that can be reaped from such support. The study presented in

chapter 5 overcomes the problem of flexibility using a hardware solution that exploits a lightweight HW monitoring system, coupled with a novel *Runtime Bandwidth Regulator* capable of dynamically controlling, in a fine-grain fashion, the bandwidth generated by the controlled accelerators.

Chapter 4

A study on Memory Interference in HeSoCs

4.1 Introduction

Designing methodologies and approaches to mitigate the memory interference problem would not be possible without a comprehensive problem characterization. An accurate experimental campaign that highlights the memory interference problem, considering different types of contention points, is of crucial importance, especially for mixed criticality systems, where missing deadlines would compromise the safety of the system. Several scientific studies are proposed in literature, that highlight the shared resource contention problem on multicore systems [54, 75, 42, 12]. However, it is necessary to go beyond the traditional multicore platforms, for which the problem is extensively investigated, extending the analysis also on heterogeneous systems. As shown in chapter 2, HeSoCs are more complex systems than traditional multicore. HeSoCs integrate heterogeneous processors and IPs, each with different bandwidth capabilities and regulation mechanisms, that are often obscure or not well documented by the hardware vendors [86, 67, 68, 66]. Heterogeneous devices also feature different interconnection architectures towards the main memory. For example, the Zynq UltraScale+ MPSoC is based on a distributed interconnect hierarchy, while the Versal ACAP is based on a NoC infrastructure in between the engines and the main memory. These architectural differences would lead to different interference sensitivity and slowdowns on the task execution time, that a system designer must take into account.

This chapter shows an in-depth analysis of the memory interference problem on two different types of HeSoCs: one GPGPU-based and two FPGA-based. Section 4.2 provides an analysis of L2-cache interference, which represents the first

level of shared memory embedded in a modern computing system (see the previous chapter), highlighting that shared accelerator memories could also be source of contention. In section 4.3 this analysis is further extended on two different generations of FPGA-based systems, namely the Zynq UltraScale+ and Versal ACAP, with a particular focus on main memory. This extends the state of the art, since the memory interference problem is less investigated on FPGA-based HeSoCs, despite they are widely adopted in the embedded and real-time domains.

4.2 Interference Analysis of Shared L2-cache on GPGPUs

Integrated GPGPUs (iGPUs) [47, 49] are still preferred to other acceleration paradigms especially, in applications with data-parallel workloads, such as computer vision and AI systems employing deep neural networks. However, the increased architectural complexity of modern iGPUs makes it extremely cumbersome to perform an effective non-pessimistic worst-case timing analysis of system. As previously introduced, the main source of unpredictability are shared resources, like the main memory, but yet only few works [23] focused on shared GPU last-level cache (LLC) which also is a major source of contention, *that affects both host and accelerator complexes*. The reason for this lack of material is that, hardware providers (in this case, NVIDIA) are too often reluctant to disclose the internals of their highly-optimized architectures and memory drivers, forcing researcher to a huge effort of reverse engineering for understanding them [23]. This is also interesting because *last-level-caches are the closest shared resources between cores*, hence they are affected by the whole memory traffic due to local caches misses, and deserve a special attention.

CUDA streams. One of the main performance booster, when adopting a host-accelerator paradigm, is the possibility of overlapping multiple computation kernels and data transfers between the CPU and the GPU. In NVIDIA GPGPUs, this is possible thanks to the abstraction of *CUDA streams*, where both execution and data transfer requests are issued from the application control running on the host. Unfortunately for RT engineers, CUDA streams introduce an additional level of parallelism, further increasing system complexity, and it is shown how the complex mechanism for stream management implemented in platform drivers *enables an additional source of contention in the system*, negatively affecting predictability, because they create interference not only on the shared memory, but also on last-level cache.

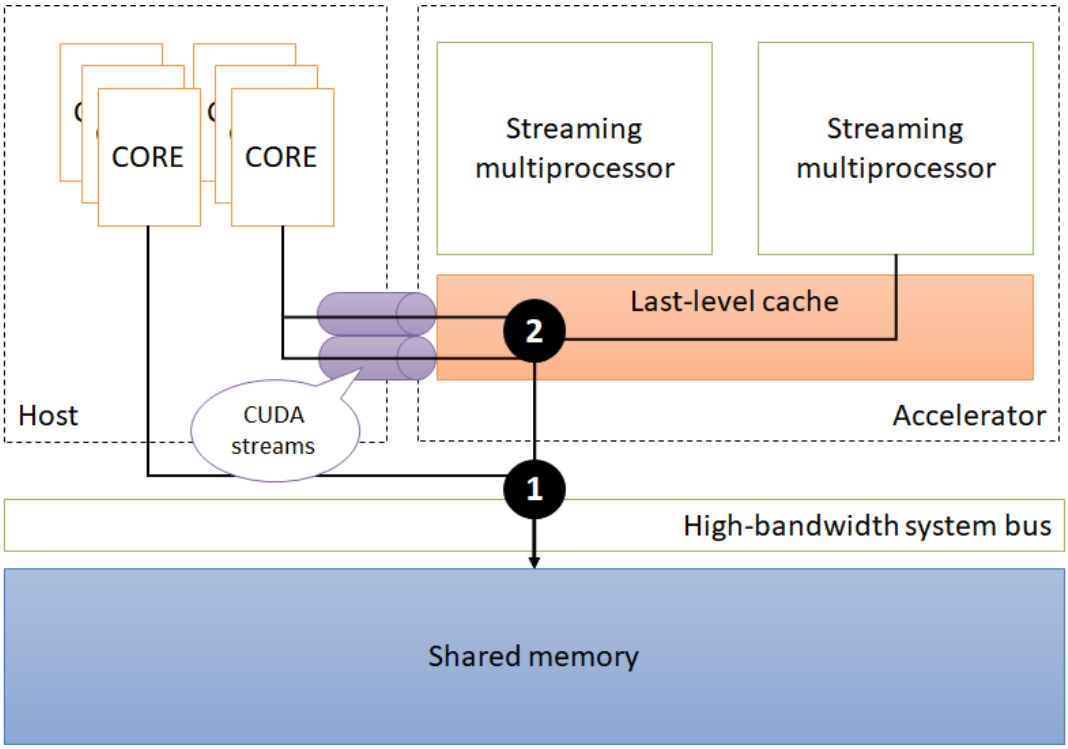


FIGURE 4.1: Reference iGPU architecture with key architectural bottlenecks

Circles with numbers in Figure 4.1 highlight the two main contention points (LLC and memory) in the considered system.

Platform modeling in industry. Another issue stems from the fact that industrial-grade frameworks for software development, such as Amalthea [3] for the automotive domain, too often rely on simplified platform model, practically inapplicable and ineffective with the complex structure of iGPUs. Indeed, there is no standard approach to model both the implicit memory contention between host cores and GPU cores. Of course, the situation gets even worse when CUDA streams are included in the picture. This work wants to be the first one in analysing and modeling, not only analytically but also with empirical evidence, the contention on LLC introduced by the adoption of multiple CUDA streams.

The rest of this section is structured as follows: in subsection 4.2.1, some ideas are provided, regarding the implementation of the interference benchmarks and some expected results. Subsection 4.2.2 shows the graphs obtained by running the experiments described in subsection 4.2.1 on top a representative embedded iGPU, the NVIDIA Tegra X2 and details the results. Finally in subsection 4.2.3 the experiments carried out are summarized.

4.2.1 Experimental Setup

In this subsection some details are provided, considering the benchmark suite used in the experiments. The slowdown due to the interference on LLC has been measured, considering two basic operations of Linear Algebra, respectively the sum of vectors and the multiplication of matrices (*vadd* and *gemm*). From the point of view of the experiments carried out, the level of cache interference is quantified by measuring the performance of the aforementioned kernels with and without interference, in terms of execution time, because unfortunately, there are no performance counters available to quantify the LLC miss-rate[1]. In this way, the execution times of the two compute-kernels with and without interference are compared. The slowdown obtained, is obviously due to the LLC replacement and the consequent high cost of access to the *Shared Memory* outside the chip. The benchmarks presented follow these two cases:

i) Compute Kernel and Interference Kernel: this first test case considers a comparison between a compute kernel running on a single *Streaming Multiprocessor (SM)* in isolation (ie. without interference), compared to the same kernel in a concurrent execution with a kernel that reads data from memory, running on a parallel stream, mapped to the second SM inside the Tegra X2 SoC.

```
1 while (runs --) {
2     idx = threadIdx.x * stride;
3     while (idx < n) {
4         w[idx] = r[idx];
5         idx    += blockDim.x;
6     }
7 }
```

LISTING 4.1: Interference kernel implementation.

Listing 4.1 shows the simple idea behind the interference kernel used in this test case. The *stride* parameter inside the code snippet, determines the access pattern of the threads to the LLC.

Figure 4.2 shows what happens inside the GPU LLC from a graphical point of view, varying the stride. In particular if it is less than the cache line size (< 32) the expected interference generated is "relatively low", because some threads will perform cache hit on the LLC. Similarly, by increasing this parameter beyond the LLC line size (> 32), a similar behaviour is highlighted, where some cache lines

[1] the metric *l2_l1_read_hit_rate* is not implemented on our target platform

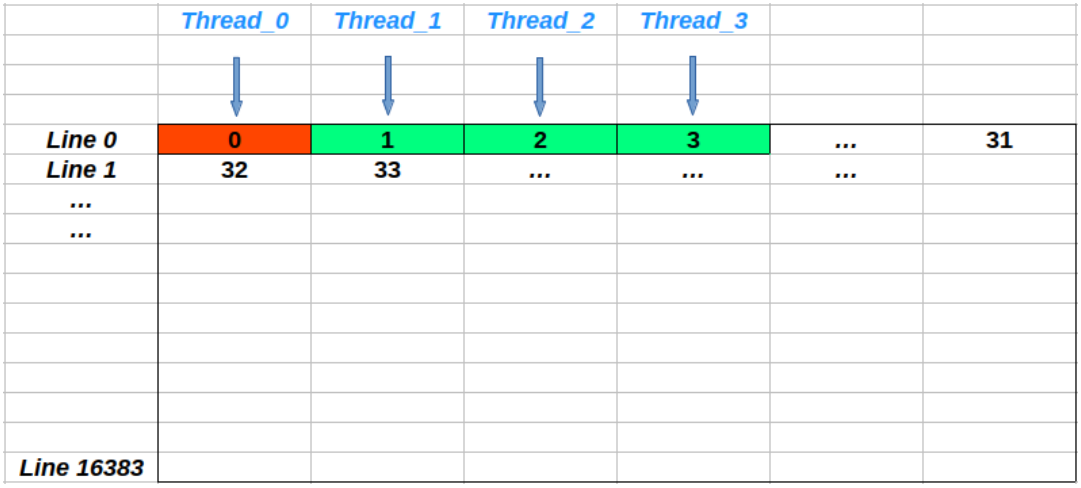


FIGURE 4.2: Thread access pattern to LLC. The case reported in this figure is a *coalesced* access, that occurs when the *stride* is equal to 1. In this case only the first thread causes a cache miss (the red cell).

will not be affected by the interference memory accesses. Finally the case of strided accesses of the same size as the cache line (= 32), generates the maximum possible interference that can be generated involving a single SM [2], then in this last case, each thread could potentially perform a cache miss, triggering a load from the main memory.

ii) Compute Kernel and Copy Engine: in this last comparison the focus is on the behaviour generated by concurrently launch a compute-kernel, running on top of one of the two SMs, together with a concurrent CUDA-stream that takes care of performing memory copy *host-to-device*, through the *Copy Engine*. This type of copy, moves the target memory, which is stored in an area addressable by the Host complex, inside a region that is visible by the GPU, and also prefetches the data needed on the GPU LLC, in order to bring this data closer to the *Compute Engines*. Within our benchmark, these memory copies involve all or some parts of cache lines, in particular the cache regions are kept under interference throughout the computation time of the concurrent kernel.

```
1 while(runs --) {
2     cudaMemcpyAsync(w_gpu, r_gpu, LINE_SIZE * cache_lines,
          cudaMemcpyHostToDevice, s1);
3     cudaStreamSynchronize(s1);
4 }
```

LISTING 4.2: Concurrent memory copy stream.

[2]The maximum number of threads that can be launched in a block is hardware-limited to 1024 threads, summing each dimension of the grid.

The benchmark reported in Listing 4.2 shows in detail the idea described above. In particular the *CudaMemcpyAsync* has been evaluated considering memory copies executed on top a concurrent stream, which is different from the default one. The *cache_line* parameter allows to control the number of cache lines under interference, ranging from one single line up to the whole cache size.

4.2.2 Experimental Results

This subsection reports the experimental results from two kernels in isolation, compared to a concurrent execution with the two aforementioned interference cases. The target computing platform is the NVIDIA Tegra X2 [49], a *System-on-Chip (SoC)* that embeds an esa-core processor, in a Big.SUPER configuration, composed of four ARMv8 Cortex A57 and two proprietary NVIDIA Denver. The GPU is composed of two Streaming Multiprocessors (SMs) belonging to the *Pascal* family, summing up 256 CUDA Cores @ 854 − 1465MHz in total. As mentioned before, one SM was used to perform useful computation and the other one for doing memory interference. Regarding the memory hierarchy, the system features 8 GB of LPDDR4, shared between the CPU cluster and the GPU, with a bus width of 128 bits and a bandwidth of 58.4 GB/s. The cache system is composed of a L2 cache, shared between the host complex and a L2 cache for the *Pascal* GPU, the second one is composed of 16384 cache lines of size 32 bytes each, summing up 512KB in total.

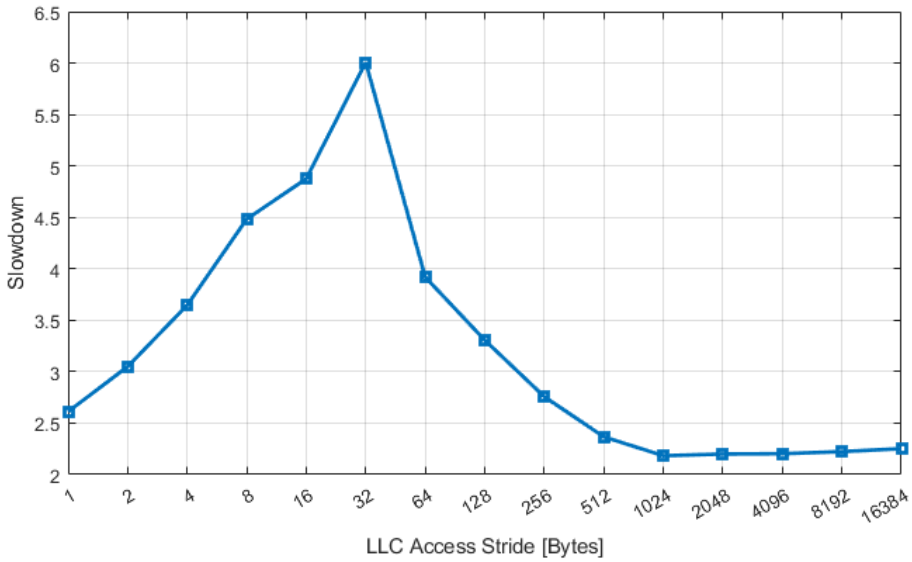


FIGURE 4.3: Slowdown of the *vadd* kernel when executed concurrently with an interference kernel.

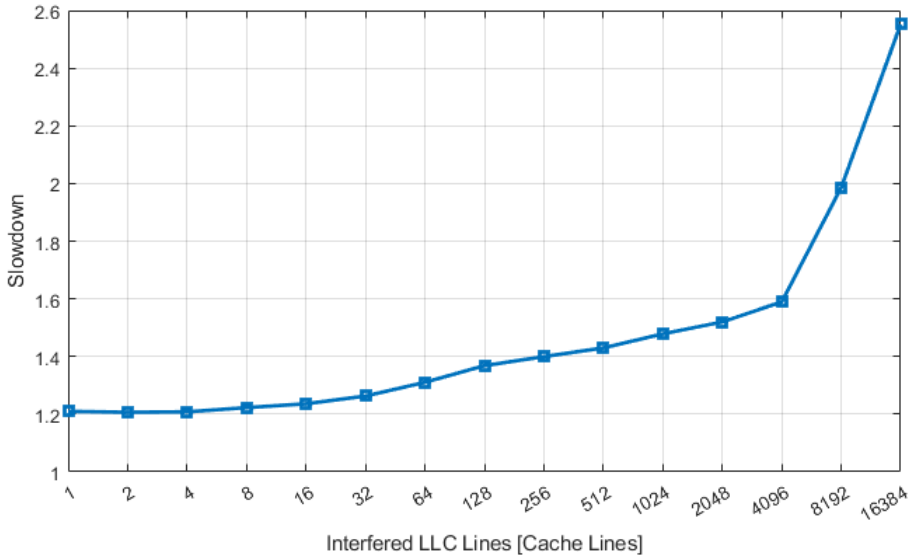


FIGURE 4.4: Slowdown of the *vadd* kernel when executed concurrently with a CUDA stream that performs memory copy *host-to-device*.

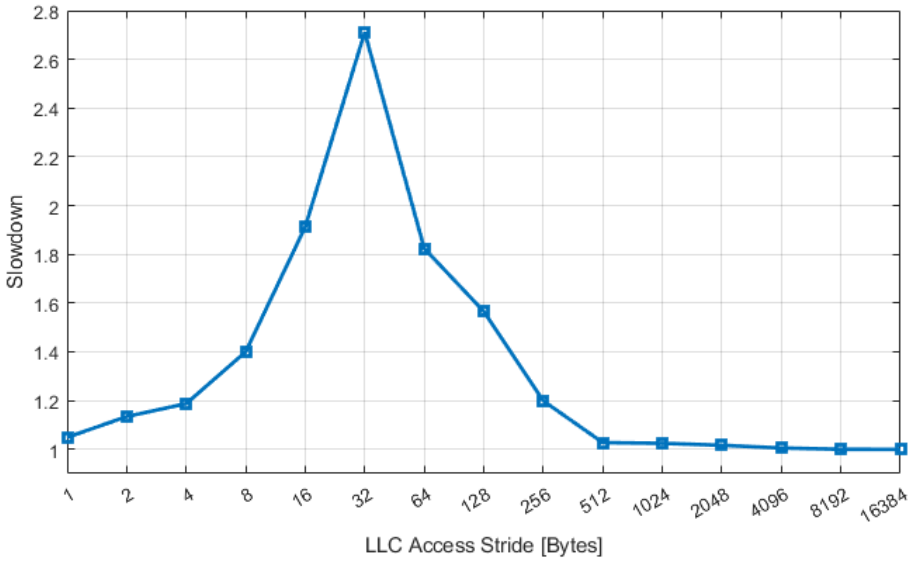


FIGURE 4.5: Slowdown of the *gemm* kernel when executed concurrently with an interference kernel.

Figure 4.3 shows the slowdown suffered by the *vadd* kernel, when the interference kernel is co-scheduled. Co-scheduling the interference kernel with the vector-add, and varying the *stride* parameter imply a linear growth in terms of execution time, until reaching the dimension of the LLC line-size. In this case the maximum number of possible cache misses and the consequent loads from the main memory that can be generated with a single SM, composed of one CUDA block of 1024 concurrent threads has been reached. The increase in execution time experimented is proportional to a 6× factor compared to the baseline version. Further increasing the

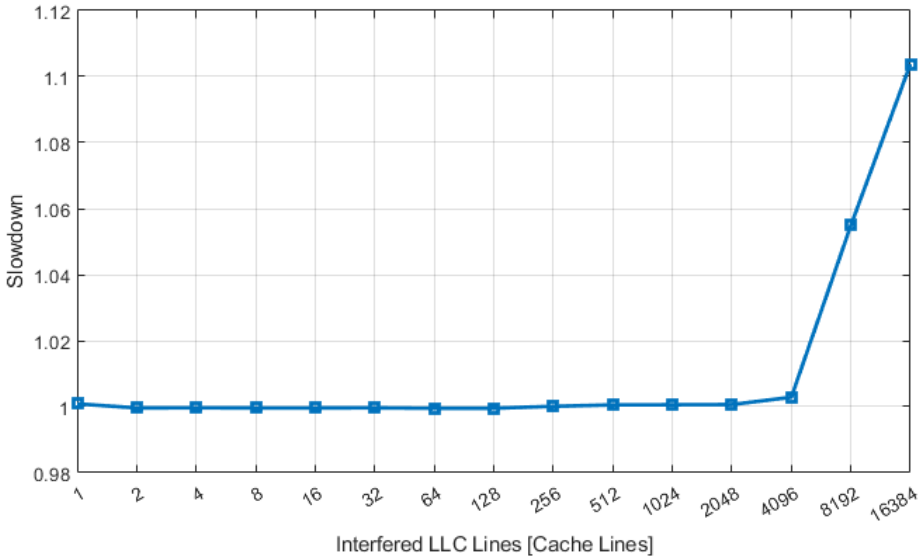


FIGURE 4.6: Slowdown of the *gemm* kernel when executed concurrently with a CUDA stream that performs memory copy *host-to-device*.

stride parameter even beyond the LLC line-size shows a reduction of the interference factor, up to about 2× compared to the case without interference. Specifically, what emerged is that the execution time seems to converge towards a specific value. The decrease in execution time is due to the fact that increasing the *stride* parameter beyond the line size of the LLC, causes that some cache lines are skipped and consequently generating less interference.

Figure 4.4 shows the behaviour of the same compute kernel, with a concurrent execution of a CUDA stream that performs memory copies *host-to-device*, exploiting the *Copy Engine*. The comparison is always measured in terms of execution time and in this case, was carried out by gradually increasing the number of cache lines affected by the memory copy stream, ranging from a single cache line, up to 16384 cache lines (the full size of the LLC). This case leads to an increase of about a 1.2× factor in the case of a single cache line interfered, up to a factor of 2.4× when the interference involves the whole cache.

The same test modalities are shown in figure 4.5 and 4.6, in particular Figure 4.5 shows the growth of terms of slowdown due to the interference generated by the concurrent kernel in execution on the second SM. Also in this case the plot shows a linear growth in terms of slowdown, by increasing the *stride* parameter, until reaching the line-size of the LLC (32-bytes). The case of the *gemm* task shows an increase of about a 3× factor of the execution time respect to the baseline reference. Beyond

this point, also in this case, follows a decrease in the total execution time, until reaching a fixed value, similar to the baseline case.

Finally for the last experiment, Figure 4.6 reports the effect of interference due to memory copy *host-to-device* does not have a particularly interference impact on the total execution time, unlike what happens in the case of concurrent kernels. It is important to highlight the difference in terms of slowdown suffered by the two considered kernels (*gemm* and *vadd*). The vector addition is sensitive by interference more than twice as the matrix multiplication. This behaviour is justified because the vector addition is a more memory-intensive benchmark than the matrix multiplication, for this reason the higher ratio of memory accesses results in a greater slowdown when the system is affected by memory interference.

4.2.3 Conclusion

This section presented, through a sequence of benchmarks, the interference effects on *Last-Level Cache (LLC)*, due to the execution of an interference kernel, mapped on a concurrent *Streaming Multiprocessor (SM)* and by means of memory copy *host-to-device*. The tests were carried out on a device that nowadays can be considered as the state of the art regarding high performance iGPU, the NVIDIA Tegra X2 SoC. Through the analysis of the results obtained it was possible to infer the size of the single cache line, a fundamental parameter to be known in the development process embedded software that makes efficient use of caches. It also emerged that interference on LLC, in case of kernels running on separate SMs, has a very high impact on the number of accesses to the main memory, which translates into *i)* a considerable increase in total execution time, of about $6\times$ in the case of the *vadd* and about $3\times$ for the *gemm*. *ii)* difficulty in making predictable the *worst-case execution time (WCET)*. This justifies and supports the need for memory interference mitigation techniques also on iGPU-based platforms, if concurrent CUDA-streams are adopted.

4.3 Interference Analysis of Shared Main Memory on FPGA-based HeSoCs

Like most high-end embedded systems, also FPGA-based ICs are increasingly adopting heterogeneous designs, where a general-purpose, multi-core host CPU is coupled to the acceleration logic on the same chip, sharing interconnect and main memory (DRAM) controller. This paradigm allows for system scalability, reduced production costs and shorter time-to-market, but creates resource contention issues, which ultimately affects the timing of the tasks deployed on the CPU or the FPGA fabric. This hinders the predictability property for which FPGAs have long been relied upon.

The Predictable Execution Model (PREM) [55] is a notable paradigm for the development of parallel software that is robust to memory interference, offering a task model where execution is structured as a repeating sequence of memory phases (that access the shared memory) and compute phases (that only uses local caches and registers). By allowing only a single actor at a time to execute its memory phase, PREM avoids timing delays due to memory interference by construction. PREM has been studied in the context of single-core CPU and I/O [55], multicore CPUs [2] and HeSoCs [22]. Although effective at guaranteeing predictable timing of memory accesses, PREM and PREM-like approaches [14, 40] greatly under-utilize memory bandwidth, which in HeSoCs is designed to concurrently serve multiple computing units. Recently, *Controlled Memory Request Injection* (CMRI) has been proposed as a technique that can operate on top of PREM-like schemes, enabling for fine-grained control of the unexploited bandwidth [16].

Most research works that try to characterize memory interference are targeted at multi-core CPUs [54, 5] or GPU-based HeSoCs [12, 80, 22], while not much has been done so far in the context of FPGA-based HeSoCs. This section aims at filling this gap by means of: i) an extensive characterization of the memory interference problem on two state-of-the-art FPGA-based HeSoCs for high-end embedded systems: the *Xilinx Zynq UltraScale+ ZU9EG* and the *Xilinx Versal VC1902 ACAP*; ii) hardware extensions to support the CMRI technique [16] suitable for generic FPGA accelerators. These hardware extensions are used in this section, to build a memory traffic regulator that will be used as an accurate main memory traffic generator to stress and assess the memory interference.

4.3.1 Accelerator Template with CMRI support

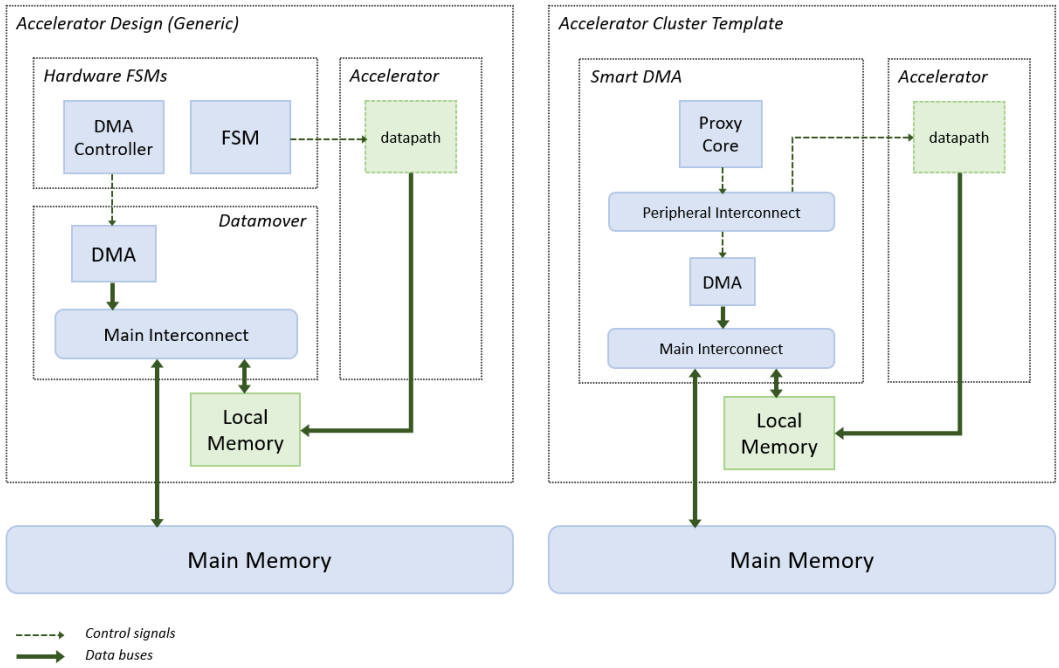


FIGURE 4.7: Two architectural designs for accelerator deployment. On the left a more traditional design, where the accelerator and DMA datapaths are controlled by an FSM. On the right the architectural design used in this work, where the control is implemented by a soft-processor.

To conduct an in-depth analysis and characterization of the memory interference in a SoC a typical methodology is that of relying on synthetic workloads aimed at stressing corner cases [12, 16]. When focusing on a FPGA-based HeSoC, a typical way of generating configurable synthetic memory traffic is that of deploying some form of *traffic generators* [43]. More in general, full-custom acceleration logic is typically designed as shown in left part of Figure 4.7, where the core acceleration logic (datapath) is coupled to some sort of *data mover* or DMA engine and a local memory. The *data mover* leverages a Finite State Machine (FSM) or DMA controller to supervise the flow of data in and out the local memory as the datapath executes.

The extensions to support Controlled Memory Request Injection (CMRI) have been built on top of this general block diagram to design. The *Accelerator Cluster Template* is shown in Figure 4.7 *(right)*. Here, the DMA and local memory is considered as immutable parts of an IP, the *Smart DMA*, that can be interfaced to different datapaths. To flexibly support different types of control logic for the DMA (and the datapath itself) this design relies on a programmable core – the *proxy core* – rather than on custom FSM logic. This paradigm has proven to be effective as moves the

control on the software side, allowing for improved flexibility and reconfigurability [6, 31]. The *Smart DMA component* allows DMA-based memory accesses in a controlled manner, which can be used to ensure that the bandwidth request generated from the FPGA accelerators does not impact the performance of other active cores/-tasks in the system beyond what can be tolerated.

Similar to what has been done by the authors that originally proposed CMRI [16], in this reference setup the most sensitive tasks to memory interference are those that run on the *host* CPUs (APU and RPU). Indeed, as introduced in Chapter 2, the Programmable Logic blocks can generate much higher sustained bandwidth request to the DRAM controller than what can be done on the ARM Application Processors. On the other hand, if one decides to stick to a PREM-like *one-at-a-time* arbitration of memory accesses and prevent completely the FPGA accelerators from accessing DRAM while the host CPU tasks are also accessing DRAM, the system bandwidth would be severely under-utilized. Given this scenario, the goal of this mechanism is that of controlling DMA requests from FPGA accelerators in a way that at any moment the maximum bandwidth that does not degrade the execution latency of the CPU tasks beyond a determined threshold. This has been achieved on multi-core CPU SoCs via *throttling* of the memory requests, which was implemented on the Smart DMA through *duty cycling*.

Listing 4.3 shows a small piece of code that highlights the main idea of the duty-cycled loop executed on the Smart-DMA component described above. Very simply, given a standard DMA transfer request to the Smart DMA, described as the triple $< \textit{destination address}, \textit{source address}, \textit{size} >$, the Proxy Core splits up this request into smaller transfer requests (*line 5*, via the `dma_memcpy` primitive) interleaved with a configurable amount of *idle cycles*. This allows a fine-grained adjustment of the bandwidth requested to the shared memory subsystem. The notion of *Cluster Load Intensity (CLI)* is defined to indicate in these experiments, the amount of *throttling* that is applied to the system. Intuitively, when IDLE_CYCLES is zero the accelerator clusters operate at full throttle, which corresponds to 100% CLI. Altering the IDLE_CYCLES parameter produces different, controllable amounts of CLI.

```
1 // dst_buffer   : address of the destination buffer
2 // src_buffer   : address of the source buffer
3 // transactions : number of transactions to serve
4 // SIZE         : size of a memory transaction
5 // subcopies    : number of copies interleaved with idleness
6 // IDLE_CYCLES  : idle cycles to interleave with memory activity
```

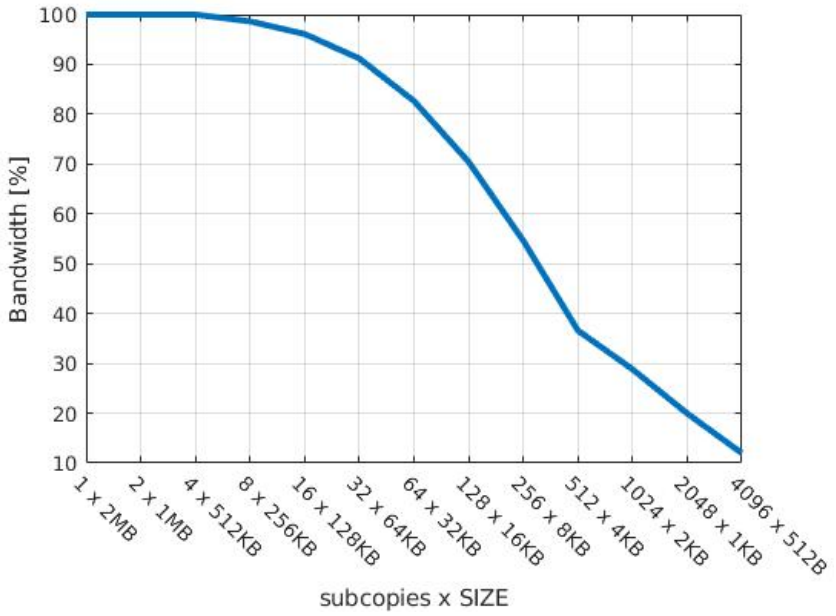


FIGURE 4.8: Tradeoff between fine-grained memory transactions and DMA programming overhead.

```
for(size_t i = 0; i < transactions; ++i){
    // a single transaction is split in
    // subcopies to interleave idleness
    for(size_t j = 0; j < subcopies; ++j) {
        dma_memcpy(&dma,
            dst_buffer+j*(SIZE/subcopies),
            src_buffer+j*(SIZE/subcopies),
            SIZE/subcopies
        );
        idle(IDLE_CYCLES);
    }
}
```

LISTING 4.3: firmware running on the *Proxy Cores* to implement CMRI.

From the implementation point of view, standard components provided with the Xilinx *IP Catalog* has been used. The *Proxy Core* is based on the *Xilinx MicroBlaze*[3]. Figure 4.8 shows the maximum amount of bandwidth that can be obtained by varying the *subcopies* parameter. The labels on the x-axis is the number of subcopies and their respective size, for example $8 \times 256KB$ indicates that a copy of `SIZE` = 2MB has

[3] https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug984-vivado-microblaze-ref.pdf

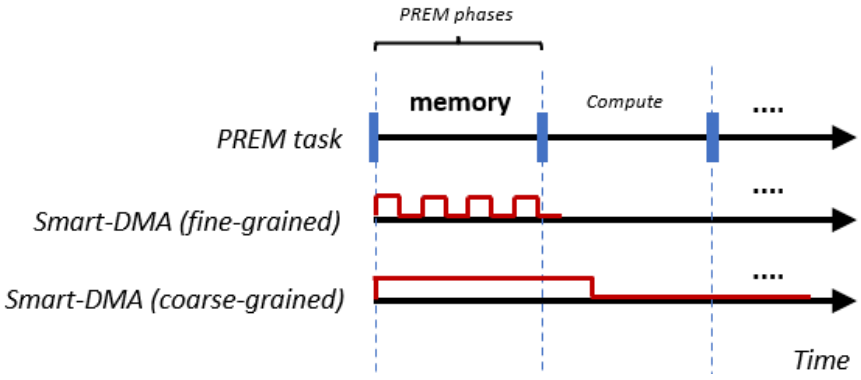


FIGURE 4.9: Impact of adopting fine-grained *subcopies*.

been split into *subcopies* = 8 of 256KB each. Recent PREM implementations consider the L2-cache as a good candidate to prefetech data [22]. The systems considered in this work feature a L2-cache size of $\approx 2MB$, for this reason, the considered size of memory transactions is `SIZE` = 2MB. Figure 4.9 shows the effect of having a fine-grained smart-DMA, i.e. able to operate even on small PREM regions. For the *fine-grained* case, it is possible to interleave memory transactions with idleness, allowing for finer bandwidth adjustments. The *coarse-grained* version operates with larger copies, which does not allow bandwidth adjustment within the memory-phase in the considered example.

4.3.2 Experimental Results

To evaluate the impact of memory contention on FPGA-based HeSoCs, a HW/SW design has been proposed. The design is composed of a multi-*accelerator cluster* system, implemented in FPGA, following the architectural template presented in subsection 4.3.1. For the ZU9EG platform, to conduct the evaluations, all the four AXI HP ports has been enabled and connected to a dedicated accelerator cluster: in this way the system can generate the maximum amount of bandwidth request from the FPGA to the DRAM controller. The same number of accelerator clusters has been instantiated on VC1902, in this case the connection to the memory subsystem takes place via the AXI NoC component. For both platforms, the system designs is closed *@300MHz*. A small control firmware is deployed on the *Proxy Core* of the *Smart DMAs*. The *Smart DMA* is capable of monitoring the CLI of the memory transactions, as discussed in subsection 4.3.1. By exploiting this code snippet, the memory bandwidth injected into the system while varying the number of clusters has been computed.

The composite memory bandwidth of the four clusters has been measured by a simple monitoring system implemented in FPGA through a small state machine, synthesized with *Vivado HLS*. This component is able to synchronize the execution of the *Smart DMA* within the clusters. When all the *Proxy Cores* are in a `READY` state, a timer is started, as soon as the last cluster ends its memory copy, an `ENDED` state is notified to the state machine, which takes care of stopping and recording the number of clock cycles took. At the end of the execution, one of these clusters identified as master, takes care of recording the obtained bandwidth value, expressed in GB/s, as:

$$bandwidth = \frac{SIZE \times DMAs \times clusters \times 1e-9}{clock_cycles \times \frac{1}{frequency}}$$

In the previous formula, the `SIZE` parameter indicates the copy size that in the tests presented is equal to the amount of BRAM configured for the two platforms. `DMAs` represents is the number of parallel transactions that are carried out by each cluster, since in these test cases the clusters are configured to perform a concurrent read and write, in this case is equal to two. In order to test a real configuration, a Linux kernel image has been compiled using the *PetaLinux* toolsuite, for tests that involve the APU of the two reference systems. The benchmarks running on Cortex R5F, has been compiled as *baremetal* applications. The PetaLinux image has been integrated with a custom *root file system* based on the *Ubuntu 20.04.2* distribution to take advantage of its rich ecosystem of software packages.

```
1  // data structure used to perform memory accesses
2  struct s {
3      struct s *ptr;
4      uint32_t dummy[(CACHE_SIZE - sizeof(struct s*))/sizeof(
           uint32_t)];
5  };
6
7  register TYPE *p = &array[0];
8  p = p->ptr;
9  // loop of memory transfers
10 while (p != &array[0]) {
11     p = p->ptr;
12 }
```

LISTING 4.4: Synthetic benchmark tested. The next element can be in a sequential or random address.

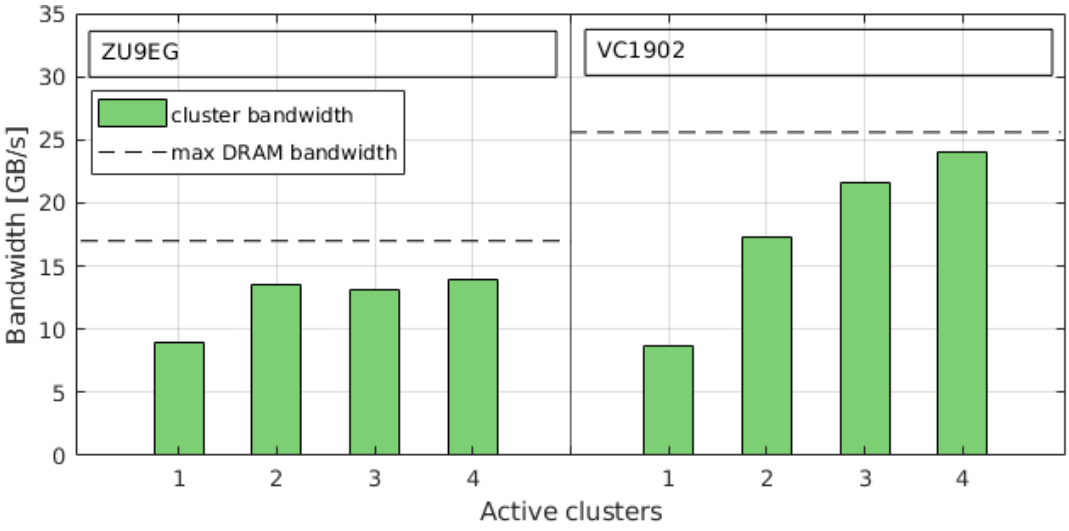
To quantify the interference on the RPU cluster, two micro benchmarks are considered, which are capable of carrying out *sequential* and *random* memory traffic patterns towards the DRAM. These two memory access patterns represent the worst case for realistic patterns that can occur in a real-life scenario. An example of pseudocode that implements these two patterns can be seen in Listing 4.4. Specifically, both benchmarks are tuned to *maximize the number of cache misses*, to ensure the issued requests are in fact serviced from the DRAM (and not intercepted by the cache hierarchy). For the *sequential* access pattern the memory reads are performed with stride equal to the L2 *Cache Line Size* which is *64B* and *32B* for the APU and RPU respectively. For the *random* access pattern the stride is randomic, but always a multiple of the cache line size. Typically this pattern exhibits a higher average miss latency, as the prefetching mechanisms in the DRAM itself (e.g., row buffers) are bypassed. The micro-benchmarks are modeled after the `lmbench` test suite[4]. As a reference benchmarks for the APU cluster, the considered real workloads are the *PolyBench* suite[5]. Also in this case the working set size (WSS) exceeds the L2-cache of the two considered SoCs. Finally as a reference workload for the AI Engines of the VC1902 ACAP, several state-of-the-art object detection and classification CNNs are evaluated, CNNs that are widely adopted on embedded platforms. Pre-trained versions of these networks are available on Xilinx Model Zoo[6].

Self interference among CPU cores

The aim of the experiments reported below is to measure the amount of self interference that the CPU tasks are affected in isolation, i.e. when no other systems (like FPGA) are issuing transactions. In Figure 4.10, the bar plots represent the maximum amount of bandwidth the FPGA can reach, by varying the number of active accelerator clusters. This amount of bandwidth is compared to the theoretical maximum for these platforms, presented in Chapter 2 and that is highlighted by the dashed line. The subplot on the left refers to ZU9EG MPSoC, where it is possible to see that this system is able to reach about $14GB/s$. This corresponds to 82% of the nominal maximum bandwidth, which is $17GB/s$. On the right, the same experiment is repeated on VC1902. In this case by enabling the four clusters the bandwidth issued reaches $24GB/s$, which corresponds approximately to 94% of the maximum

[4] http://lmbench.sourceforge.net/
[5] https://web.cse.ohio-state.edu/pouchet.2/software/polybench
[6] https://github.com/Xilinx/Vitis-AI/tree/master/models/AI-Model-Zoo

FIGURE 4.10: Accelerator Clusters



nominal bandwidth supported by the system, which is 25.6GB/s. These two experiments show that on ZU9EG the maximum bandwidth saturation has already been reached with two clusters, in fact by adding more clusters the amount of bandwidth remains unchanged. From the point of view of the VC1902, the amount of bandwidth scales almost linearly with the number of clusters. Intuitively by adding more clusters the memory bandwidth will reach a saturation point as for the ZU9EG platform. Another important aspect that emerges from this experiment concerns the memory bandwidth repartition between FPGA and Processing System. The same experiment shows that the FPGA itself can not saturate the entire memory bandwidth, but a small fraction of the bandwidth is reserved for the host cores. The tests shown in Figure 4.11 measure the maximum DRAM bandwidth available on the RPU and APU of the two systems. The graphs show that in both systems the maximum memory bandwidth the *host* can reach is $\approx 7GB/s$ for sequential memory access pattern. This test also highlights two different architectural choices adopted in the two reference systems. In the ZU9EG MPSoC a single core from both the APU or RPU complex only uses a fraction of the total bandwidth, and an almost additive behaviour is observed as more cores are used, highlighting that no self interference occurs with this configuration. On the VC1902 system the same applies to the RPU, but on the APU a single core can saturate the available memory bandwidth to the whole multi-core CPU. When the second core is activated, this total bandwidth is evenly shared between the two cores. In terms of absolute values, the RPU generates an order of magnitude smaller peak memory bandwidth request compared to

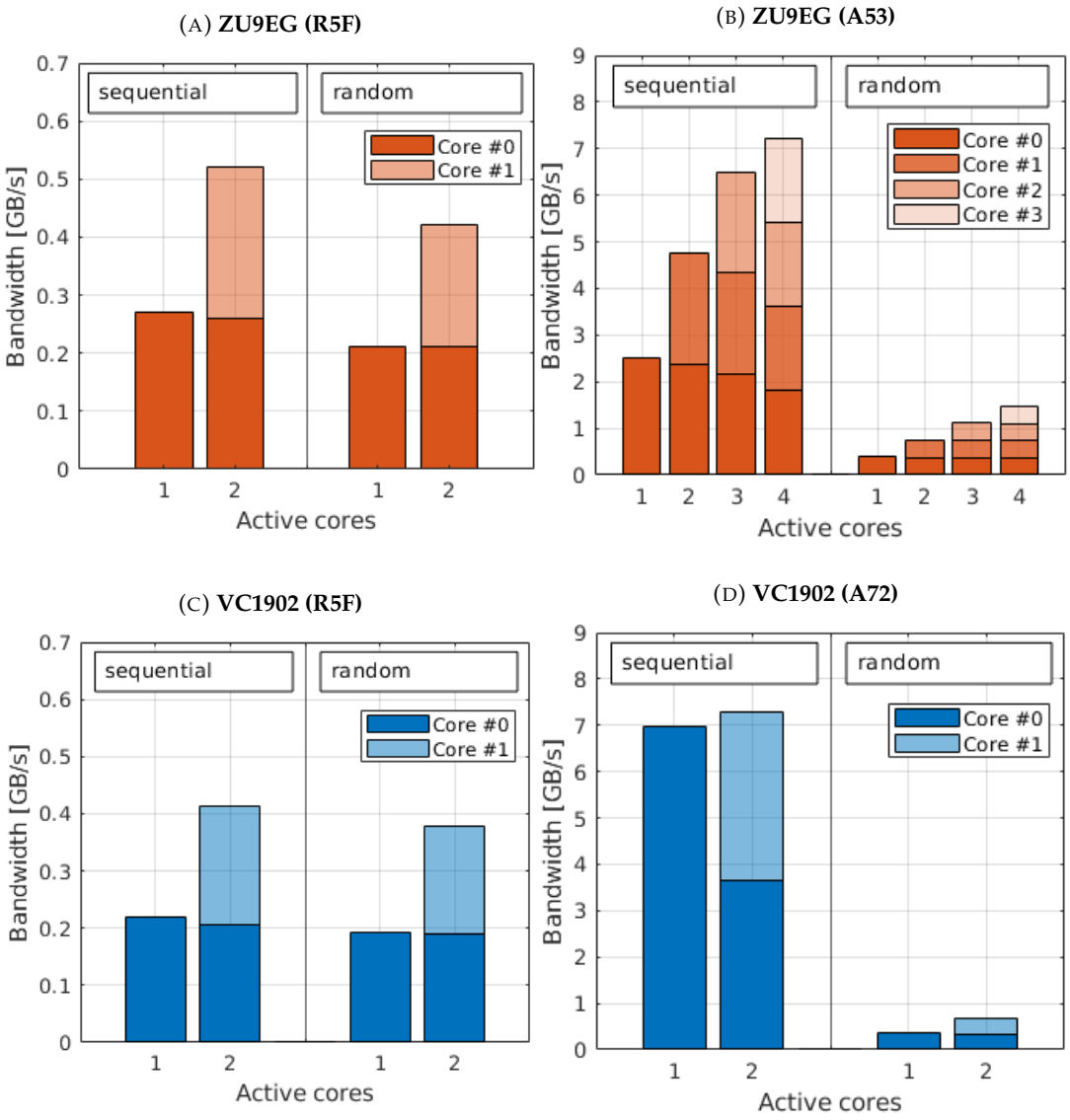


FIGURE 4.11: Comparison of the DRAM bandwidth available on the host cores of the two systems. (4.11a) the results obtained on R5F and (4.11b) the same test on A72.

the APU, and there is barely any difference between the two memory access patterns. This is due to the low operation frequency of the RPU cluster.

To carry out the experiments reported in this section, micro-benchmarks are adopted and configured to generate a sequence of L2-cache misses, therefore most of the memory accesses occur directly in DRAM, as previously explained. The memory accesses identified as sequential follow a pattern with a fixed offset set at 64Bytes, which corresponds to the the L2-cache line size, for the systems considered. This results in a cache-line fill at each memory access. While representing the worst case, a sequential or strided access pattern is often applied on regular data structures, such

as vectors or matrices, widely adopted for linear algebra operations. A random access pattern, on the other hand, is often applied to algorithms and data structures that make extensive use of pointer chasing.

Memory Interference among heterogeneous engines

The following experiments aim to quantify the impact of memory interference on a workload running on one of the *host* cores. This can be referred as workload *under test*. Specifically the following scenarios are considered:

1. The task under test is a computational kernel part of the popular PolyBench benchmark suite. Different types of kernels are proposed, including: linear algebra primitives, solvers and statistical operations. In this case the test runs on the application processors (A53 and A72). These benchmarks have been configured in such a way that their Working Set Size (WSS) exceeds the L2 cache, thus maximizing the number of cache misses generated. This is representative of the worst-case.

2. The task *under test* is one of the synthetic benchmarks running on a single core from the RPU processors of the considered SoCs. In this case the interference is generated both by the remaining cores in the RPU cluster (also running *random* and *sequential* synthetic benchmarks) and by the accelerator clusters in the FPGA;

3. **[VC1902 only]** The task *under test* is one of several Neural Network workloads running on the AI Engine. In this case the interference is generated by the accelerator clusters on the FPGA.

The idea is to exploit the Accelerator Cluster Template (ACT), presented in this section to study and analyze the problem of memory interference. In this case, the fine-grained bandwidth regulation implemented within the Smart-DMA is used as a configurable traffic generator. As the goal is to study the worst-case scenario, the system is composed of four active clusters. The x-axis of the graphs shown from Figure 4.12 to 4.23 reports the CLI sweep, which indicates the memory activity of the clusters controlled by interleaving memory copies and idleness as explained in this chapter. For example, CLI = 80% indicates the operating point where four clusters are at the same time issuing memory requests with $80\% - 20\%$ copy-idleness duty cycle. The y-axis of these plots indicates the *slowdown* suffered by the task under test.

1) Interference on real workloads on APUs The curves on the graphs refer to the number of actives cores within the system. Since the ZU9EG platform embeds a quad-cores A53 processor on the ZU9EG there are four curves. For VC1902, only two curves are shown as the A72 processor is equipped with only two cores. Figure 4.12 to 4.23 show the interference experiments related to the first case considered, which involves the interference experienced on the application cores during the execution of PolyBench kernels. In this case each group of two figures refers to a single computational benchmark. The left figure represents the experiment performed on the older generation platform (the ZU9EG), while the right figure represents the VC1902 SoC.

General Considerations The proposed experiments highlight that in general the execution time strongly depends on the behaviour of the other tasks deployed on the system, due to memory interference effects. Like the study proposed in the previous section, this first of all justifies the need for bandwidth regulation schemes and mitigation techniques also for FPGA-based systems. In general, the new VC1902 platform, as shown in Figure 4.10 allows the FPGA to reach a maximum higher bandwidth rather than ZU9EG, therefore in this first case the interference experienced by the cores is greater. This highlights that this problem is relevant in particular for the VC1902 platform. As evidenced by the experiments, the maximum interference factor measured on VC1902 is equal to 16$\times$ for the `fdtd-2d` kernel (Figure 4.17b) and $\approx 8\times$ for the `durbin benchmark` on the ZU9EG platform (Figure 4.21a).

Saturation Points The graphs highlight a plateau at $\approx$ 50% CLI for the ZU9EG platform. As shown in Figure 4.10, two clusters with CLI = 100% are enough to reach the bandwidth saturation for the ZU9EG platform, this justifies the observed behaviour. On the other hand, on the new VC1902 platform, a higher amount of bandwidth is available than the previous generation of SoCs. The saturation point, in this case, is reached as soon as all four clusters considered for the experiment are activated simultaneously. The saturation plateau on the memory interference would therefore be visible in case of activation of more than four concurrent clusters. However it was not possible to carry out experiments considering more than four clusters due to the reduced amount of Block RAM available on the two reference devices.

(A) **ZU9EG (A53)** (B) **VC1902 (A72)**

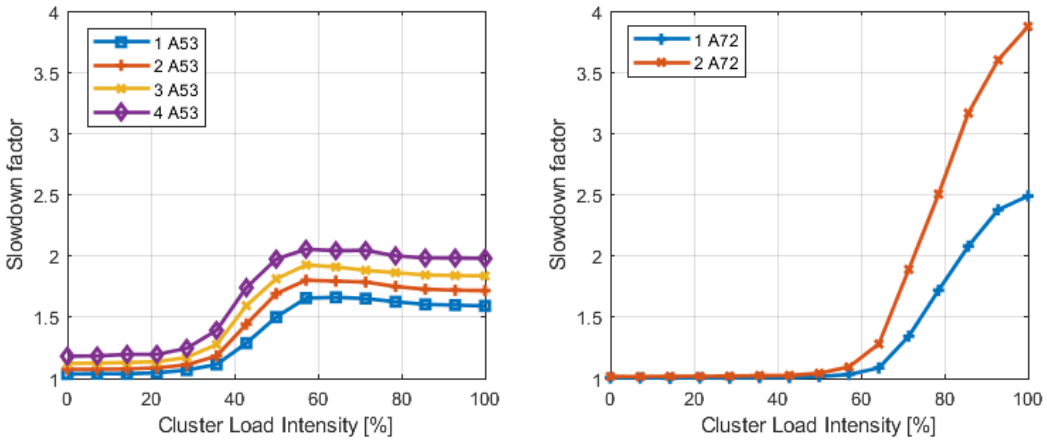


FIGURE 4.12: Alternating Direction Implicit solver (ADI)

(A) **ZU9EG (A53)** (B) **VC1902 (A72)**

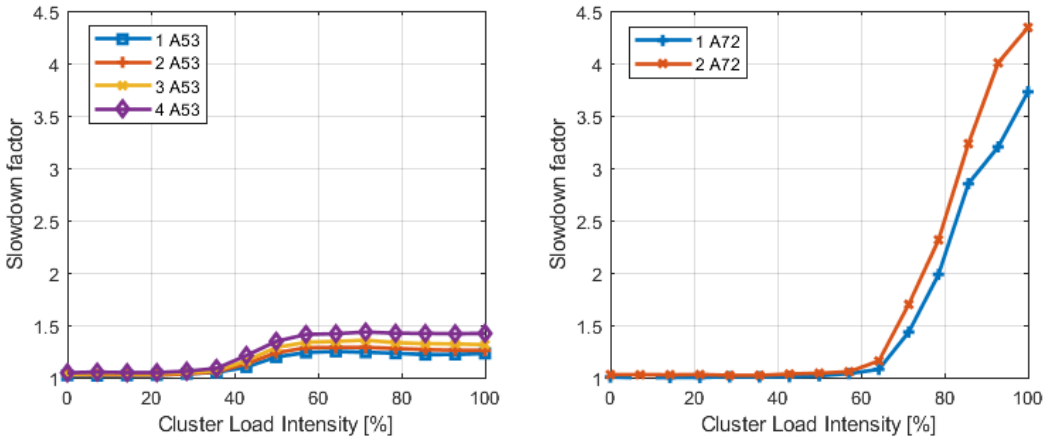


FIGURE 4.13: Matrix Transpose and Vector Multiplication (ATAX)

(A) **ZU9EG (A53)** (B) **VC1902 (A72)**

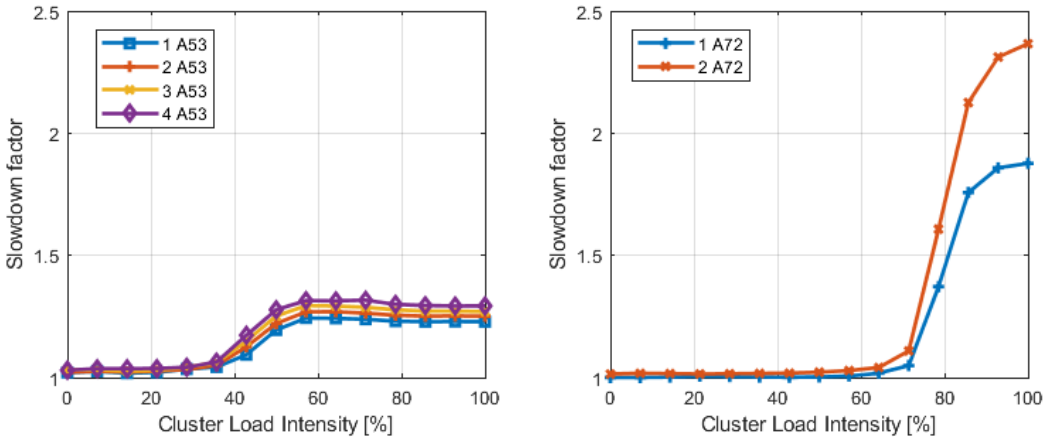


FIGURE 4.14: BiCG Sub Kernel of BiCGStab Linear Solver (BICG)

(A) **ZU9EG (A53)** (B) **VC1902 (A72)**

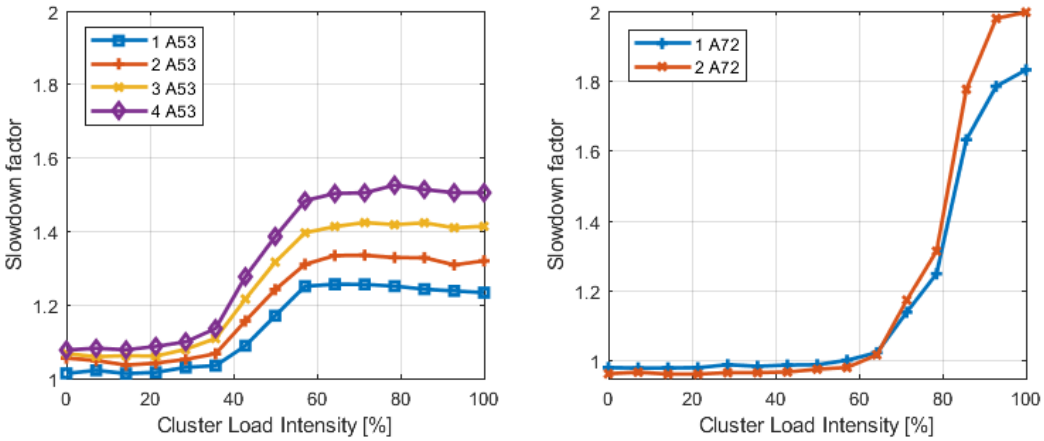


FIGURE 4.15: 2-D Convolution (Conv2D)

(A) **ZU9EG (A53)** (B) **VC1902 (A72)**

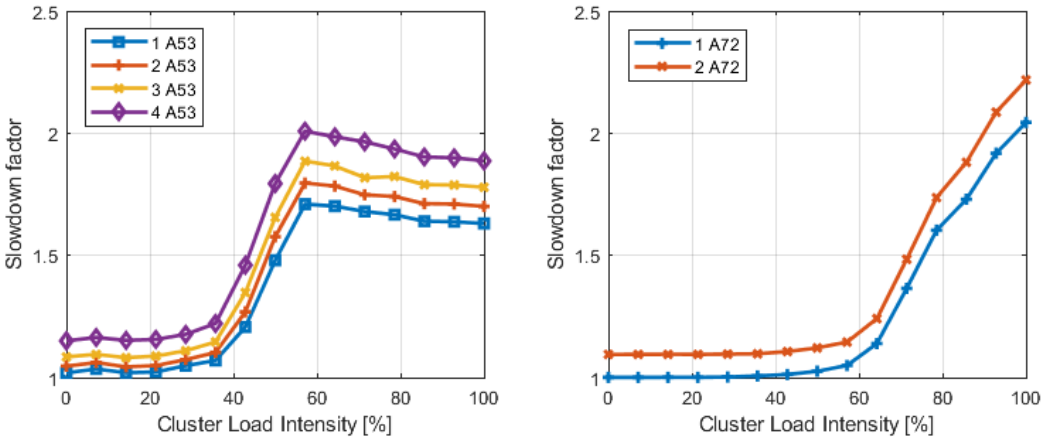


FIGURE 4.16: Covariance Computation

(A) **ZU9EG (A53)** (B) **VC1902 (A72)**

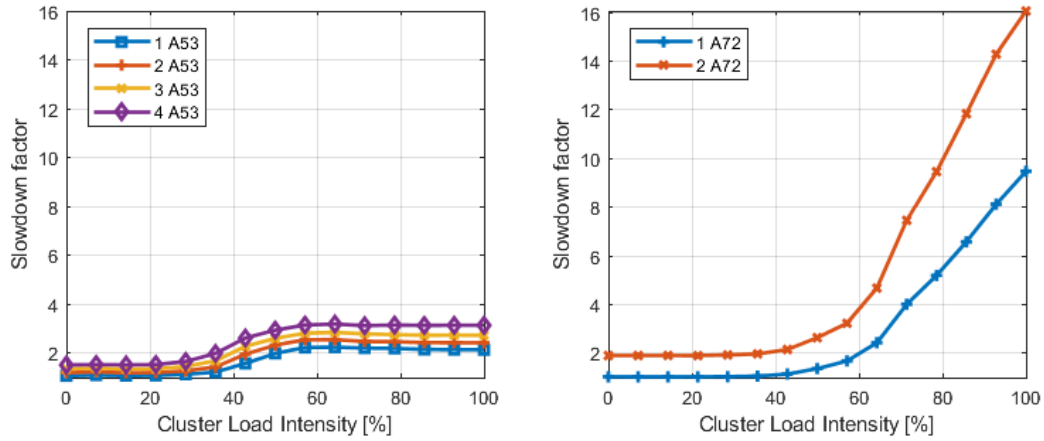


FIGURE 4.17: 2-D Finite Different Time Domain Kernel (FDTD-2D)

(A) **ZU9EG (A53)** (B) **VC1902 (A72)**

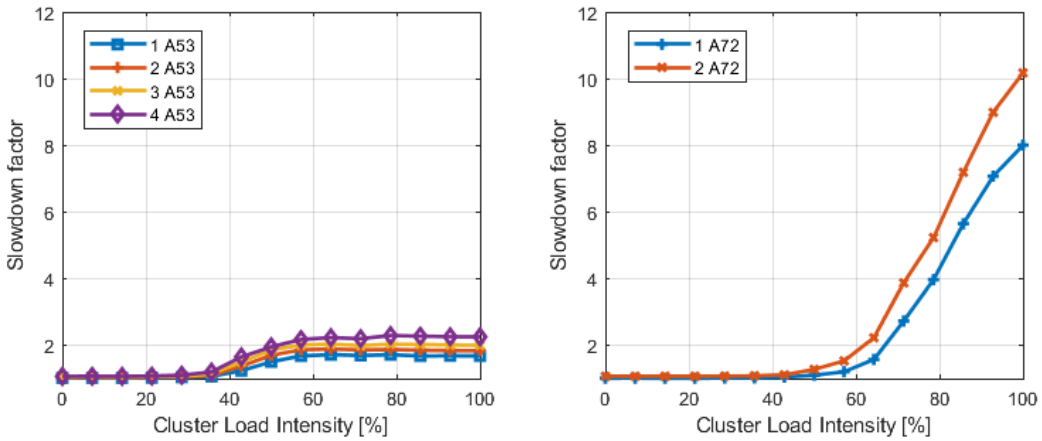


FIGURE 4.18: Cholesky Decomposition

(A) **ZU9EG (A53)** (B) **VC1902 (A72)**

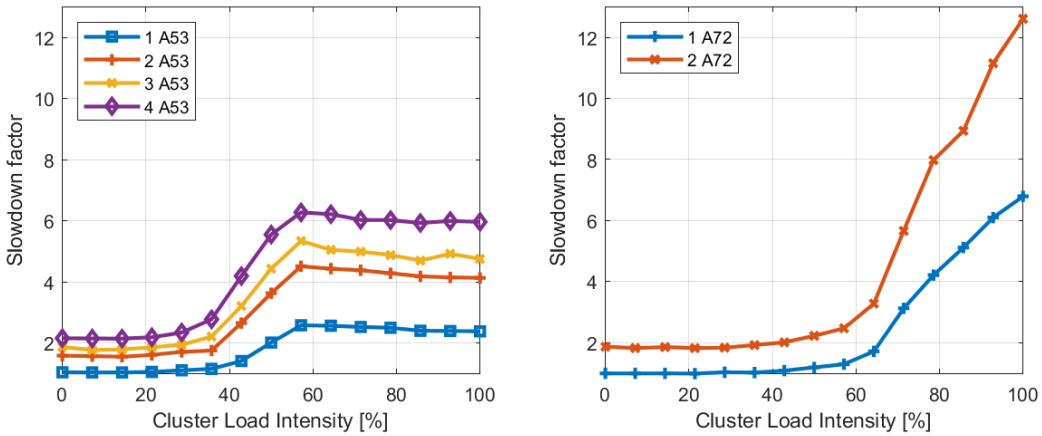


FIGURE 4.19: Matrix Vector Product and Transpose (MVT)

(A) **ZU9EG (A53)** (B) **VC1902 (A72)**

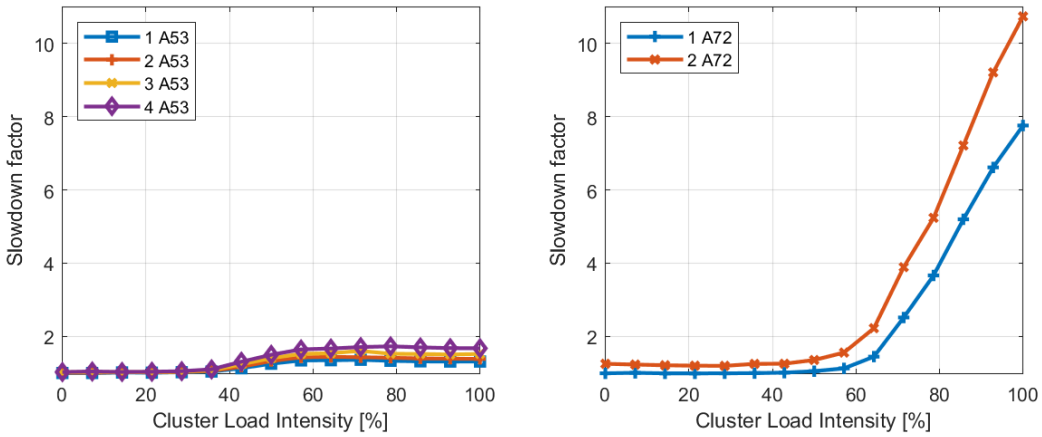


FIGURE 4.20: Triangular solver (TriSolv)

(A) **ZU9EG (A53)** (B) **VC1902 (A72)**

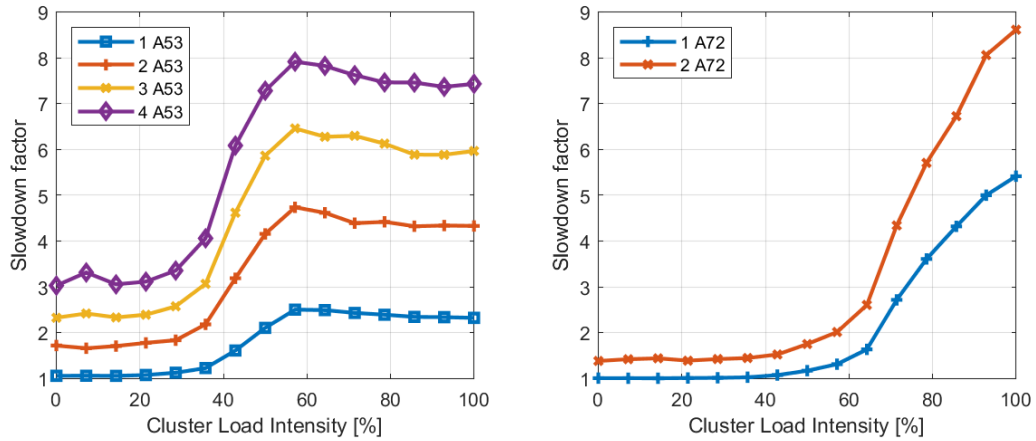


FIGURE 4.21: Toeplitz system solver (Durbin)

(A) **ZU9EG (A53)** (B) **VC1902 (A72)**

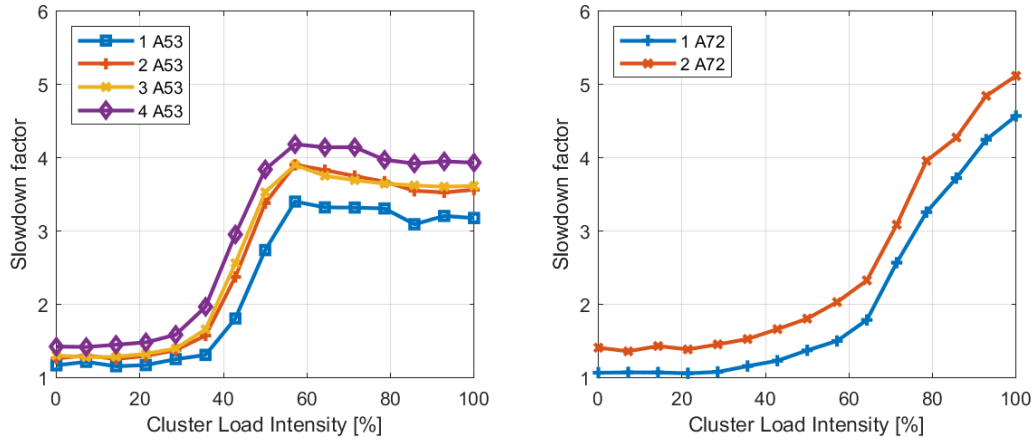


FIGURE 4.22: Gram-Schmidt decomposition

(A) **ZU9EG (A53)** (B) **VC1902 (A72)**

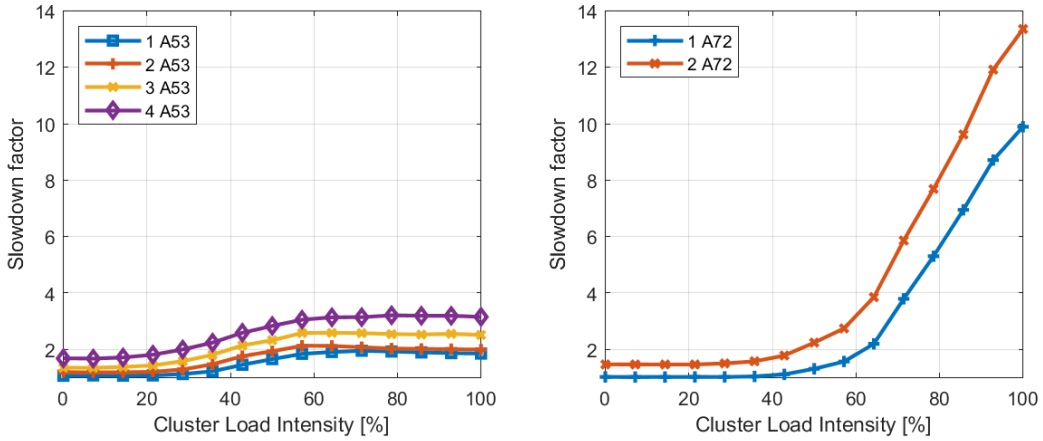


FIGURE 4.23: LU decomposition

Inter-core interference The behaviour observed in this evaluation highlight another trend that worth to be mentioned. Specifically, two possible different behaviours can be identified: i) the inter-cores interference (the distance between the curves), depends on the memory activity of the FPGA clusters (from the CLI parameter); ii) the inter-cores interference does not depend on the activity of the clusters. This first category includes the benchmarks shown in Figure 4.12b, 4.13b, 4.14b, 4.15b, 4.18b. In this case the inter core interference becomes visible when $CLI > \approx 60\%$. This effect corresponds to the case where the cores alone would not be able to saturate the memory bandwidth, in fact when the CLI value is low, the effect of the interference is not visible. As soon as the FPGA exceeds the 60% threshold, the amount of bandwidth required exceeds the maximum available, thus generating the interference phenomena visible in the graphs. This behaviour is more evident on the VC1902 platform, where the bandwidth fraction that a single A72 core can exploit is greater than the cases reported for the ZU9EG platform.

2) Interference on Synthetic benchmarks on RPUs Figures 4.24 and 4.25 show the interference experiments carried out on the real-time cores of the two platforms considered. The first case involves the interference in case of *sequential* memory access pattern, while in the second one the access pattern is *random*. In general, the memory interference measured on the RPU cores, as expected, is less than the amount generated on the application cores. This is mainly due, as shown in Figure 4.11 to a lower bandwidth available toward the main memory, compared to the application cores. The bandwidth reduction is due to two different aspects: i) The working frequency of the RPU cluster is lower than the frequency of the APU cluster. The maximum frequency of the real-time cores is 500MHz, while the frequency of the APU cores used on these two platforms is fixed to 1.2GHz; ii) The RPU cluster can exploit a single port towards the DRAM, in the configuration considered in these experiments. In fact, the traffic is routed by crossing the *LPD Main Switch*, whose data buses are 64 bits wide. On the contrary, the APU can make use of two different ports to access the DRAM, whose bus width is 128 bits wide [7].

The memory interference measured on the two reference SoCs, in the case of synthetic benchmarks, is very similar, as the R5F cluster is the same in the two platforms. The VC1902 SoC, however, still exhibits a higher level of interference, due to

[7] the outgoing memory traffic generated by the RPU cluster can be routed to the two 128-bit ports used by the APU cluster, however this possibility has not been explored and will be subject to future investigations.

(A) **ZU9EG (R5F)** (B) **VC1902 (R5F)**

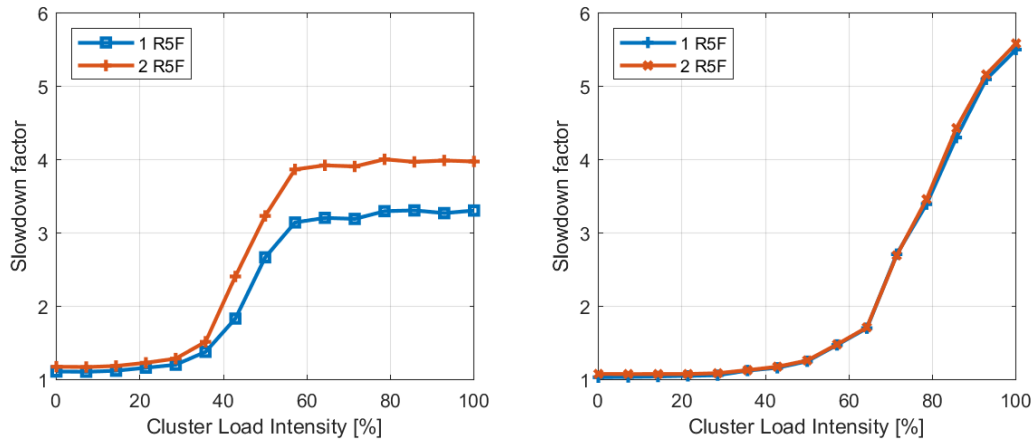


FIGURE 4.24: Sequential memory access pattern

(A) **ZU9EG (R5F)** (B) **VC1902 (R5F)**

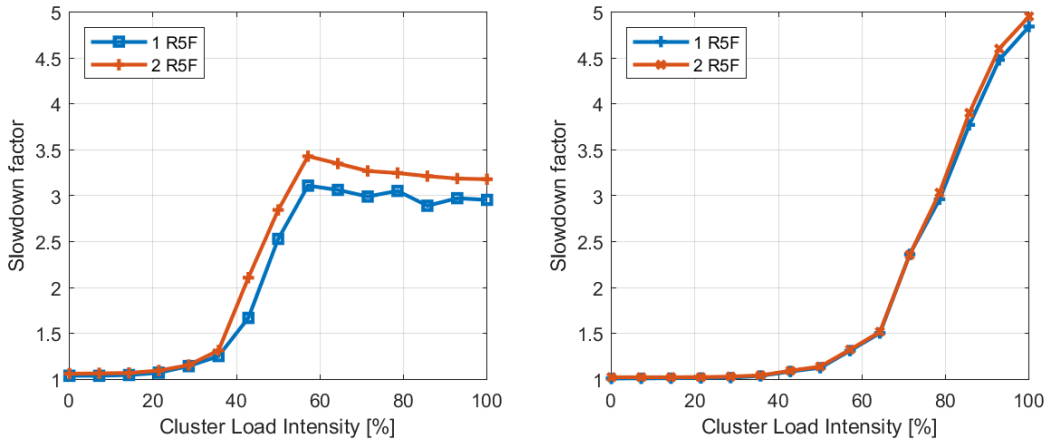


FIGURE 4.25: Random memory access pattern

a higher maximum bandwidth achieved by the FPGA cluster than the ZU9EG chip.

Compared to the computational benchmarks discussed above, there is no significant inter core interference. For example on the ZU9EG platform, considering the points where CLI < 40%, the concurrent execution of the two real-time cores can take place without significant slowdowns and only when CLI > 40% the inter core interference begins to be visible. Conversely, on the VC1902 platform, there is no inter core interference, even for CLI close to 100%. As final remark, it is important to remember that these experiments are representative of the worst-case, as the benchmarks consider continuous memory accesses without interleave computation.

3) Interference on AI-Engines accelerated Neural Networks Finally, Figure 4.26 shows the memory interference measured where the tasks under test are some neural networks running on the AI Engine neural accelerator. In this case, only the VC1902 chip was taken into consideration, as it is the only one that integrates this type of accelerator. AI Engines is a dataflow accelerator, designed for the efficient execution of algorithms that can be represented as *Direct Acyclic Graphs (DAGs)*, where the cores that are embedded inside the accelerator are interconnected with each other through a mesh topology. From the architectural point of view, this accelerator makes use of the FPGA to access the main memory, therefore the maximum amount of memory bandwidth the AIE can exploit is the same as the accelerator clusters shown in Figure 4.10. To conduct the experimental activity, some popular neural networks are evaluated, typically employed in the field of the embedded computer vision, including different versions of the YOLO models [59]. The implementation of the CNNs was achieved by means of pre-trained models available within the Xilinx Model Zoo.

The two graphs shown in Figure 4.26 refer to the case of a single neural network running on the AI Engine accelerator (left figure) and a case where two neural networks runs in parallel (right figure), in order to ensure that the systolic accelerator resources are fully occupied (and the generated bandwidth is maximized). In the second case, there are also two A53 cores that transfer the images to be computed to the AI Engines accelerator.

Different neural networks have diverse sensitivities to memory interference: while the more sensitive CNN, `refinedet_baseline`, suffers up to $\approx 2.5\times$ of slowdown, the less sensitive ones (`yolov2_voc_pruned` and `resnet50`) experience less than the $1.5\times$ slowdown even in the worst case. These differences are due to different *computation to communication ratio* of the neural networks considered, the ones that are more affected by interference spend more time in accessing the main memory rather than computing and for this reason their execution time is more dependent by the co-running tasks. The qualitative evaluation of memory interference reported in this chapter led to the following observations:

Observation 1: *The problem of memory interference can affect task predictability in a different way, on the different types of high-end embedded systems considered and could lead in the worst case, to a slowdown on the execution time up to* $16\times$*;*

Observation 2: *Due to the architectural characteristics, the memory interference problem is even worse on the new VC1902 platform compared to the previous generation of SoCs;*

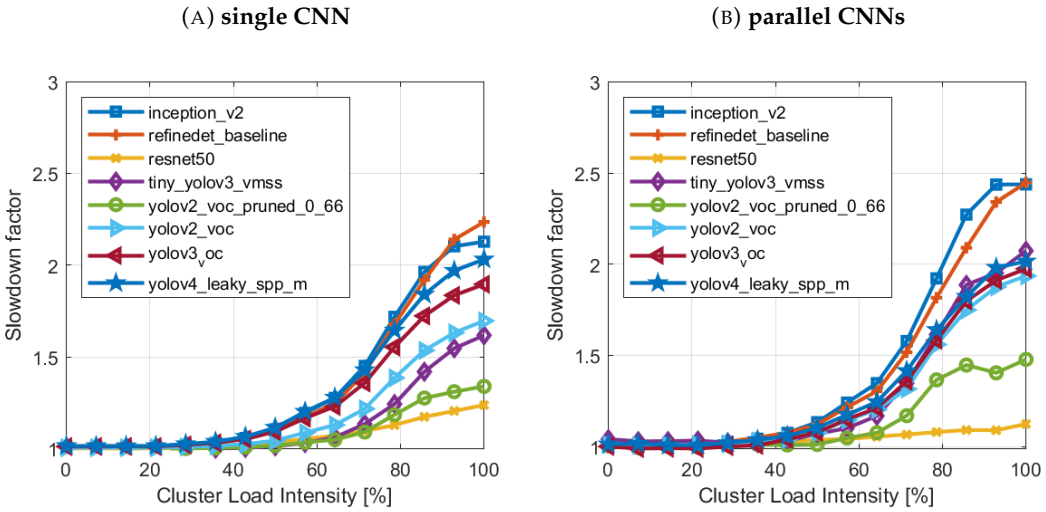


FIGURE 4.26: Convolutional Neural Networks on Versal AI Engine

Observation 3: *The RPU cluster, that is typically designed for real-time workloads, is also affected by memory interference, despite being marginally affected rather than the application cores.*

4.3.3 Conclusions

This section presented an in-depth memory interference analysis for two last generation FPGA-based HeSoCs belonging to the *Zynq UltraScale+* and *Versal ACAP* families. This fills a gap in the literature, which mostly focuses on multi-core CPUs and GPU-based HeSoCs, and confirms the fact that memory interference strongly affects also FPGA-based devices. When all the CPU cores and the FPGA logic are executing in parallel memory interference can slow down APU tasks by more than 16×, RPU cores by up to 5.58× and the AI Engines on the VC1902 by up to ≈ 2.5×. These analysis highlights thus that Real-Time RPU cores are potentially impacted by memory interference, which must be taken into account. Globally, the analysis points out that also in FPGA-based HeSoCs PREM-like approaches that arbitrate accesses to DRAM in a mutually exclusive manner are bound to severely under-utilize the available memory bandwidth.

This chapter presented a methodology to support state-of-the-art CMRI technique on FPGA HeSoCs. The bandwidth regulator is integrated in a generic template for FPGA-based accelerator clusters, based on a smart DMA. The presented approach allows a fine-grained memory bandwidth regulation, through which a study

of memory interference on two different systems has been presented. The capabilities of this approach to mitigate the interference problem will be discussed in the next chapter. The results reported in this chapter are originally presented in a paper titled *"Understanding and Mitigating Memory Interference in FPGA-based HeSoCs"* [9] and submitted at *Design, Automation and Test in Europe (DATE) 2022* conference.

Chapter 5

The Proposed Mechanism to Mitigate Memory Interference

5.1 Introduction

As explained in the previous chapters, as the number of on-chip compute engines grows, on COTS heterogeneous systems, the interference due to main interconnect and memory sharing [14] significantly hampers the tasks' execution time, making the system unpredictable from the point of view of timing. Several solutions have been proposed to tackle this problem, ranging from static memory partitioning techniques [25] [84], to task execution models that guarantee predictable memory access [55] and memory bandwidth regulation strategies (e.g., [21, 61, 83]). The latter in particular are widely available also in commercial products, based on the combination of bandwidth *monitoring* and *throttling* mechanisms. When considering heterogeneous systems based on programmable logic (FPGA), the availability of such mechanisms is typically limited to loosely coupled, coarse-grained components from the point of view of the actuation interval (i.e., the time to monitor and throttle the bandwidth). When co-scheduling numerous HW and SW tasks from real-time operating systems featuring scheduling ticks below the millisecond boundary, fine-grained and tightly-coupled schemes to support bandwidth regulation is mandatory to guarantee quality-of-service (QoS) control. While some fine-grained techniques exist, they are highly platform-dependent, addressing the problem only for specific system-on-chips [53].

This chapter presents *an innovative, fine-grained and platform-independent Runtime Bandwidth Regulator (RBR) for disciplined main memory bandwidth usage in commercial-off-the-shelf, FPGA-based HeSoCs*. The RBR, based on the CMRI extensions introduced in the previous chapter, performs tightly-coupled monitoring and throttling of main

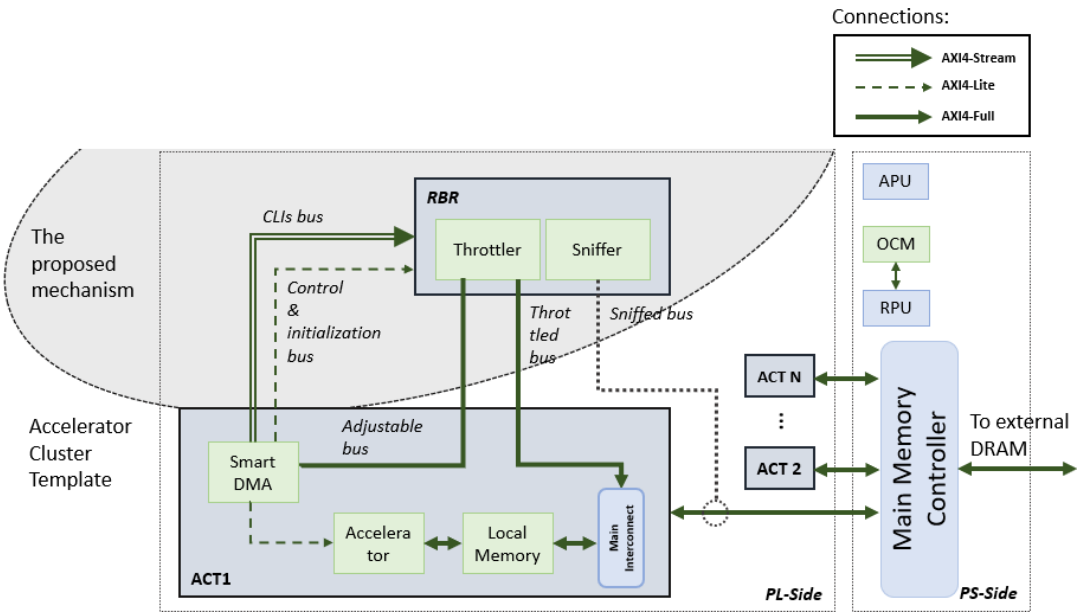


FIGURE 5.1: FPGA-based HeSoC Accelerator Cluster Template.

memory bandwidth, effectively delivering the desired QoS levels with high precision. The proposed RBR quickly adapts to dynamically varying QoS levels, and is completely platform independent. Moreover, our system does not interfere with existing tasks running on the multiple compute engines (APU+RPU), and introduces a minimal timing overhead, i.e., it takes only 1 clock cycle to perform the monitoring and throttling operation.

The mechanisms and ideas proposed in this chapter are the result of a teamwork, carried out in collaboration with the *University of L'Aquila*, in the context of the *FRACTAL*[1] and *COMP4DRONES*[2] european projects. Our experimental results show that the proposed RBR is able to precisely track and adapt to a dynamically evolving QoS request profile with a frequency 12× higher than what can be done with state-of-the-art, loosely-coupled solutions on a Xilinx Zynq UltraScale+ HeSoCs. When comparing to SW-based monitoring and throttling solutions present in the literature, our mechanism outperform them by 2 orders of magnitude. Finally, evaluations that our mechanism enables co-existence of multiple heterogeneous HW/SW tasks with tight performance constraints, will be presented.

5.2 The Proposed Mechanism

The proposed mechanism aims at regulating the bandwidth to access the main shared memory through a Runtime Bandwidth Regulator (RBR), composed of a Sniffer and a Throttler.

Fig. 5.1 shows the proposed RBR inside an ACT implemented on a Xilinx Zynq Ultrascale+ MPSoC, a commercial off-the-shelf (COTS) platform representing an embodiment of the template presented in subsection 4.3.1 of the previous chapter. The RBR takes as input four buses: the *sniffed bus*, the *adjustable bus*, the *control & initialization bus*, and the *CLIs bus* (where *CLIs* stands for *Cluster Load Intensities*), and provides one output, the *throttled bus*. The RBR, by means of the Throttler, throttles the bandwidth (for readings and writings) of the *adjustable bus* by a variable factor, depending on the activity monitored by the Sniffer on the *sniffed bus* and some *CLIs* received from *CLIs bus*. In particular, the value of *CLI* indicates the amount of throttling to be applied to the system; *CLI* can range from 0 to 1, without assuming extremes values (i.e., $CLI \in (0,1)$). A *CLI* of 0.1 means that the bus bandwidth will be reduced by 90%, while a *CLI* of 0.9 means that the bus bandwidth will be reduced by 10% (both with respect to the current value). The internals of the RBR are reported in Fig. 5.4. In the following description, it will be presented only the readings path inside the RBR (namely, the path to monitor and throttle readings bandwidth), since the writings path is the same. Moreover, the clock signals are not reported in Fig. 5.4, since all the blocks work in the same clock domain.

5.2.1 Monitoring

The Sniffer component (see Fig. 5.4) measures the latency to transfer a given number of bytes on the *sniffed bus*, and provides it to the Throttler component. The Sniffer first counts, through a Counter UP block, the number of read bytes on the *sniffed Bus*. The counter output is then sent to an equality Comparator block that sets an *idle_valid* signal high when the number of read bytes reaches a *threshold*. The *threshold* is stored in a Controller block containing a register bank, that is also in charge of coordinating Sniffer activities. The *idle_valid* signal also resets the Counter UP block. Every time there is a *reset* shot, the counter is able to activate a Timer block (through an *act* signal), that starts its free evolution and provides the latency on the *copy_cycles* output: the timer is activated at the first byte read, and continues evolving until a

[1] https://fractal-project.eu/
[2] https://www.comp4drones.eu/

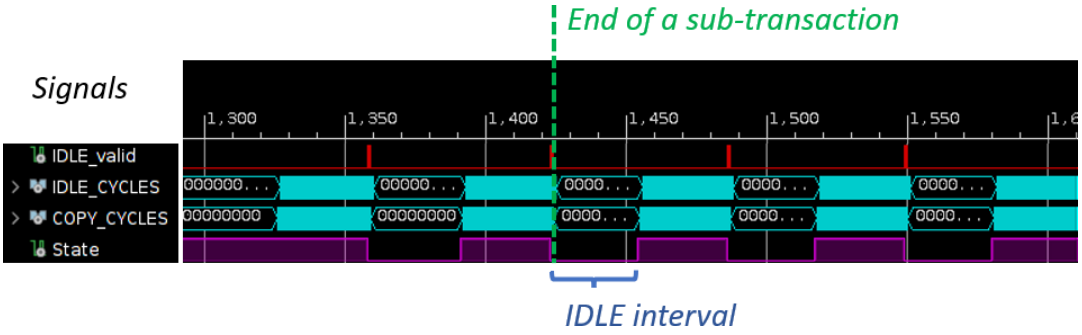


FIGURE 5.2: The RBR internal signals. `state` is the enable signal, that goes low when the sniffer sends an `IDLE_valid`. `IDLE_CYCLES` is the output of the weigher IP, based on CLI and `COPY_CYCLES`.

reset shot happens. The *threshold* value is programmable and is provided through the *control & initialization bus*. Supposing that the *i-th* byte transferred on the *sniffed bus* allows reaching the *threshold*, the total time between the flowing of the *i-th* byte and the raising of the *idle_valid* signal is 1 clock cycle. As implementation of the bandwidth monitoring component, a state-of-the-art solution proposed by G.Valente et al. [77] has been adapted and integrated inside the RBR.

The bandwidth monitoring activity is triggered by the Proxy-Core by means of the APIs reported in Listing 5.1.

```
1   // initialize the Sniffer component
2   void jointer_initialize(uint32_t * base_mon, uint32_t num);
3
4   // set threshold for the read and write AXI channels
5   void jointer_thresh_rd(uint32_t * base_mon, uint32_t num,
        uint32_t threshold);
6   void jointer_thresh_wr(uint32_t * base_mon, uint32_t num,
        uint32_t threshold);
7
8   // starts the Sniffer component
9   void jointer_run_thr_r (uint32_t * base_mon, uint32_t num);
10  void jointer_run_thr_w (uint32_t * base_mon, uint32_t num);
11  void jointer_run_thr_rw(uint32_t * base_mon, uint32_t num);
12
13  // deactivates the Sniffer component
14  void jointer_stop(uint32_t * base_mon, uint32_t num);
```

LISTING 5.1: APIs used by the Proxy-Core to interact with the Sniffer component.

`jointer_initialize` is used to initialize the sniffer component at the beginning of the Proxy-Core application. The two *thresholds* (respectively for readings and writings) can be configured by means of `jointer_thresh_rd` and `jointer_thresh_wr` APIs. After the *thresholds* has been configured, the monitoring activity can be activated by calling `jointer_run_thr_*`, based on the transaction type (read, write or read/write). Finally `jointer_stop` can be used to release the Sniffer component.

5.2.2 Throttling

The Throttler component (see Fig. 5.4) receives the latency value from the Sniffer and, depending on the *CLIs* received from the *CLIs bus*, stops the communication on *adjustable bus* for a given time. It is worth noting that a *weight* value is computed based on CLI, by the Proxy-Core as: $(1 - CLI)/CLI$. Before passing this value to the Weigher component, the Proxy-Core converts this floating point value into a fixed point representation, in order to reduce the usage of programmable logic, as reported in Listing 5.2.

```
1  uint32_t compute_weight_from_cli (float cli) {
2      float tmp = ((1 - cli) / cli);
3      tmp = tmp * (1 << 9);
4      return (uint32_t)(tmp + 0.5);
5  }
```

LISTING 5.2: weight computation based on the CLI value

The Weigher block of the Throttler component multiplies the *weight* value (which is obtained by the *CLIs*) and *copy_cycles* values, providing *idle_cycles* as output; it is worth noting that it can optionally contain a FIFO queue to store multiple *weights*. The resulting *idle_cycles* value is loaded inside a Counter DOWN block every time there is a rising-edge of *idle_valid*: the loading is activated by a finite-state machine (FSM) block through a *load* signal. The FSM is notified of *idle-valid* rising-edge by a dedicated Detector block. After loading the counter, the FSM enables, through an *en* signal, the decrementing counting. When the counter reaches 0 value, an equality Comparator block raises an *allow* signal. The *allow* signal enters a Braker block that takes as input the *adjustable bus* and provides it as output properly throttled; in particular, when the *allow* signal is low, the Braker completely interrupts the communication on *adjustable bus*. When *allow* is high, the *adjustable bus* is left untouched. When the Throttler receives the rising-edge of *idle_valid* signal, it takes 1 clock cycle to stop the communication on *adjustable bus* (the time to load the Counter DWN

block). It is worth noting that the minimum time interruption for the adjustable bus is 2 clock cycles. The close interaction between Sniffer and Throttler guarantees a **tightly-coupling** between the monitoring and throttling operations. In turn, that provides a better control of the bandwidth, that can be appreciated through an example of application. The interaction between the Proxy-Core and the DMA engine occurs with the custom made APIs listed in Listing 5.3.

```
1  // data structure that holds the status of the dma engine
2  typedef struct dma {
3      uint32_t       _dma_id;
4      uint64_t   _current_id;
5      uint32_t _rst_baseaddr;
6      uint32_t _dma_baseaddr;
7  } dma_t;
8
9  // functions that can be used to reset the DMA hardware and
       to initialize the dma_t structure
10 void dma_reset(dma_t * dma);
11 void dma_init (dma_t * dma, uint32_t dma_id, uint32_t
       rst_baseaddr, uint32_t dma_baseaddr);
12
13 // function that can be used to wait that a transaction,
       identified by the id parameter, is completed
14 void dma_wait (dma_t * dma, uint64_t id);
15
16 // to start a dma transaction
17 uint64_t dma_memcpy(dma_t * dma, uint32_t src, uint32_t dst,
       uint32_t size, uint32_t weight);
```

LISTING 5.3: APIs and data structures that can be exploited by the Proxy-Core to configure the DMA engine.

The `dma_t` structure is used to keep the status of the DMA engine, that can be reset and initialized by calling `dma_reset` and `dma_init` respectively. The `dma_memcpy` can be used to trigger a DMA transaction, enforcing a custom bandwidth requirement, using the *weight* parameter, which is computed based on CLI by using the `compute_weight_from_cli` function previously explained. Finally `dma_wait` is used to synchronize the execution with respect to the memory transaction identified by the `id` parameter.

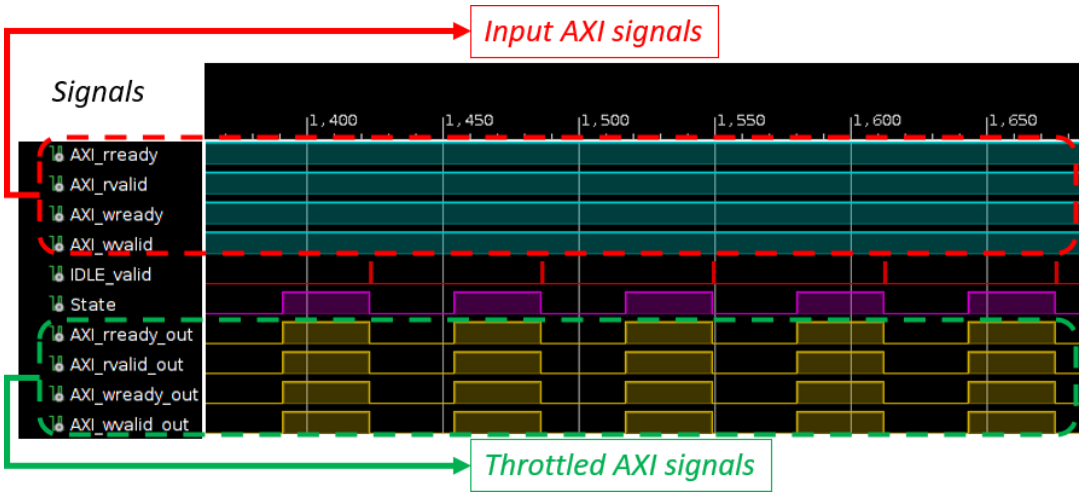


FIGURE 5.3: The internal signals of the throttler component. The AXI4 handshake signals (xvalid/xready) are masked based on the `state` of the throttler.

5.2.3 Example application

To this end, it is discussed an example of application of the proposed RBR with reference to the Fig. 5.1. The *sniffed bus* is the AMBA AXI4 bus [3] that connects the slave Main Memory Controller to the Main Interconnect of the ACT1 (please note that the AMBA AXI4 is referred as AXI4-Full in Fig. 5.1 to distinguish from AMBA AXI4-Lite). The *CLIs bus* is an AMBA AXI4-Stream bus. The *control & initialization bus* is an AMBA AXI4-Lite bus (the AXI Interconnect is not reported in Fig. 5.1). Finally, the *adjustable bus* is the AMBA AXI4 bus that connects the master Smart DMA to the Main Interconnect of ACT1: the bus is entered in the RBR and output as *throttled bus*. It is worth noting that the interruption of the AMBA AXI4 inside the Braker takes place in compliance with AXI4 specifications. During a task executed by *ACT1*, the Smart DMA first initializes the RBR with a *threshold* of 512 bytes and 1024 bytes for readings and writings, respectively. Then, to properly throttle the bandwidth, it sends a *CLI* of 0.3 and 0.5 for readings and writings bandwidth, respectively. Now the Smart DMA requires a reading transfer from memory of 600 KBs size: data starts flowing from external DRAM to Local Memory block. Once the 1st byte flowed on the *sniffed bus*, the Sniffer activates its internal Timer. Then, when the 512th byte flows, the Sniffer provides the latency to the Throttler (with a rising-edge on *idle_valid* signal). The Throttler then multiplies the reading latency for a *CLI* of 0.3 and stops the adjustable bus for the resulting time. At the end of *idle_time*, the Throttle reopens communication on adjustable bus. The monitor and throttle loop

[3] https://developer.arm.com/architectures/system-architectures/amba/amba-4

happens every 512 bytes, hence 1200 times for the 600 KBs transfer. After finishing transferring bytes to Local Memory, the Accelerator starts executing the task and writing back results to Local Memory. At some point, the Smart DMA sends 200 KB of data result from Local Memory to external DRAM: in the same way of readings, during the writing cycle the RBR applies the control in 1 clock cycle every time the 1024th byte flows on the bus, hence 200 times for the complete writing transfer. To better understand how the proposed system works, Listing 5.4 shows an example of code, running on the Proxy-Core of the Smart-DMA, that implements this simple use case.

```
void run_example_application() {

    dma_t dma_r, dma_w;

    // initialize dma engines
    dma_init(&dma_r, DMA_ID_0, RST_BASEADDR, DMA_ID_0_BASEADDR);
    dma_init(&dma_w, DMA_ID_1, RST_BASEADDR, DMA_ID_1_BASEADDR);

    volatile u32 *base_monitor = (volatile u32*) MON_BASEADDR;

    u64 id_r, id_w;

    // compute weights based on CLI
    u32 weight_r = compute_weight_from_cli (0.3);
    u32 weight_w = compute_weight_from_cli (0.5);

    jointer_initialize(base_monitor, 0);

    // set thresholds for reading and writings
    jointer_thresh_rd(base_monitor, JOINTER_0,  512);
    jointer_thresh_wr(base_monitor, JOINTER_0, 1024);

    // starts the sniffer
    jointer_run_thr_rw(base_monitor, JOINTER_0);

    // starts the transactions
    id_r = dma_memcpy(&dma_r, SRC_1, DST_1, 600*1024, weight_r);
    id_w = dma_memcpy(&dma_w, SRC_2, DST_2, 200*1024, weight_w);

```

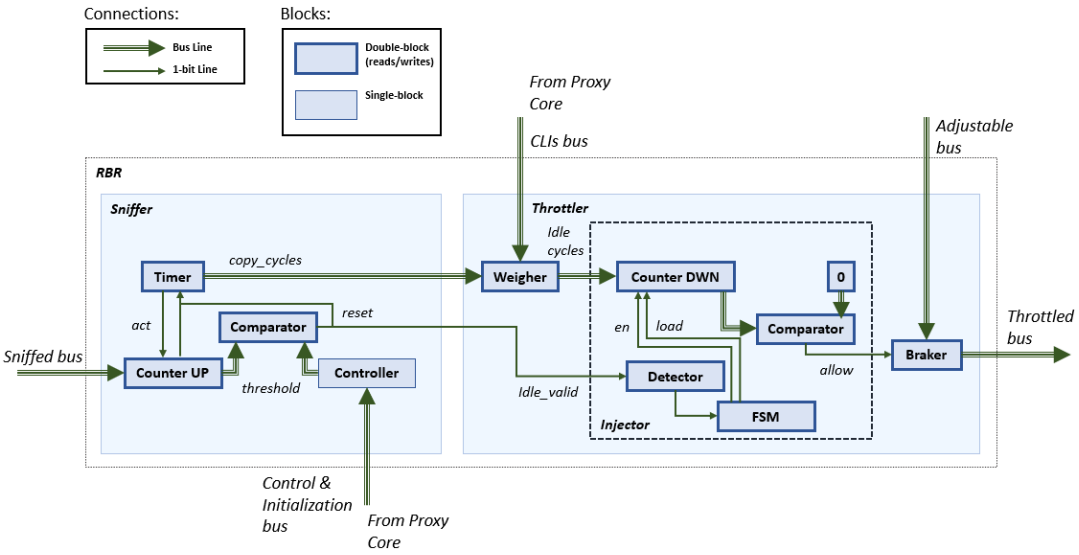


FIGURE 5.4: Runtime Bandwidth Regulator (RBR).

```
30      // synchronize the execution
31      dma_wait(&dma_r, id_r);
32      dma_wait(&dma_w, id_w);
33
34      jointer_stop(base_monitor, JOINTER_0);
35  }
```

LISTING 5.4: example of application of the proposed RBR.

Summing-up, the proposed RBR works in 1 clock cycle thanks to the tightly-coupling between monitoring (Sniffer) and throttling (Throttler) actions, leaving the frequency of the bandwidth regulation up to the set *threshold* value. This allows a more **fine-grained** actuation interval compared to loosely-coupled solutions. Also, the RBR enforces a **platform independency** because it has been developed starting from the reference architecture reported in the previous chapter, general-enough to describe an HeSoCs; indeed, a porting toward another architecture only requires to adapt the Braker block inside the Throttler and the Counter UP inside the Sniffer to match the bus type. Finally, the proposed RBR is flexible enough in shaping the bandwidth at run-time, thanks to the programmability of the CLIs. See Figures 5.2 and 5.3 as a real example extracted using the *Xilinx Chipscope IP*. From the point of view of the programmable area needed to implement our RBR, Table 5.1 shows a breakdown of the components of our design. The programmable resources used to deploy our methodology are 323 Flip Flops and 449 Lookup Tables, that are less than 1% of the total resources available on the ZU9EG MPSoC.

TABLE 5.1: Programmable area needed to implement our RBR design.

Component	Flip-Flops (FFs)	Lookup Tables (LUTs)
sniffer	134	279
throttler	189	170
RBR	323	449

5.3 Experimental Results and SOA Comparison

The goal of our tightly-coupled bandwidth monitoring and throttling (TCMT) solution is to enable an accurate bandwidth regulation on the accelerators deployed on an FPGA-based heterogeneous SoC, with minimal overheads and minimal impact on the timing. In such systems, co-operation between the main CPU and the accelerators is key to achieving performance, and it will be responsibility of the operating system to co-schedule SW and HW tasks such that their main memory usage is not subject to massive interference and uncontrolled timing degradation. In real time operating systems the scheduler tick can go below 1ms and as low as 100us [4]. In such a setting new tasks can be admitted in the system very frequently; at the same points in time co-scheduling decisions related to maximum allowed bandwidth can be re-evaluated. Consequently, the mechanism for bandwidth monitoring and throttling needs to be as responsive and tight as possible.

The experimental section is organized in four distinct parts. Subsection 5.3.1 presents an evaluation of the benefits that can be achieved by using CMRI to build a memory bandwidth regulator. Here the objective is to assess the benefits that such a technique allows in terms of memory interference mitigation. To be more general, the evaluation is proposed for both Zynq UltraScale+ MPSoC and Versal ACAP. Subsection 5.3.2 aims at assessing the cost of the proposed technique compared to standard HW/SW mechanisms available in the target platform for bandwidth monitoring and throttling. Subsection 5.3.3 evaluates the effectiveness of the proposed tightly-coupled hardware solution, compared to more loosely-coupled schemes where monitoring is done via software-controlled performance monitors. The last experiment, presented in subsection 5.3.4, highlights the benefits that can be obtained using a fine-grained bandwidth regulation on the other active CPU tasks.

[4] https://github.com/evidence/erika3

5.3.1 Benefits of CMRI technique

The experiments presented below aim at studying to which extent it is possible to allow concurrent execution of FPGA accelerators and CPU or AIE workloads – provided that it is possible to control and bound the effects of interference – as opposed to PREM-like DRAM-arbitration schemes, where only one task at a time is allowed to access the main memory. In particular, similar to the original CMRI setup [16], the measurements involve the memory bandwidth that the throttled entity (here, the FPGA accelerators) can utilize while maintaining the slowdown imposed on the tasks *under test* below a given threshold. Two thresholds are considered, at 1.1× and 1.2× slowdown, and the (CLI), i.e., the maximum injection rate that does not exceed such thresholds, is shown. Clearly these threshold values are only for illustration purposes: different applications/systems will have different maximum tolerated latency increase.

Memory interference mitigation analysis

Note that in this context CLI acts as an indicator of FPGA bandwidth usage efficiency (BWE), as it represents the bandwidth injected by the FPGA cluster(s) normalized to the maximum bandwidth they can request (see Figure 4.10). Like in the previous experiments, these two cases are considered: i) where the task *under test* runs on the RPU/APU and ii) the task *under test* runs on the AIE. The setup in terms of benchmarks and configurations is the same. Figure 5.5 shows the results for this experiment, where a single FPGA cluster is injecting traffic at various CLI rates. The plots in this figure show workload slowdown with increasing CLI. The markers with circular and rhomboidal ends highlight the ranges in which CMRI operates within the tolerated 10% and 20% thresholds, respectively.

Focusing on the APUs as units under test (Figures 5.5a and 5.5b), it is possible to see that CMRI allows to exploit 22-to-43% of the maximum bandwidth that a single cluster can generate on the ZU9EG (8.93GB/s) and 28-to-72% on the VC1902 (8.68GB/s), while guaranteeing a maximum latency increase of the workload under test of up to 10%. If the tolerance threshold is increased to 20% CMRI can exploit 23-to-53% of the maximum cluster bandwidth on the ZU9EG and 40-to-78% on the VC1902. Note that if the same experiment is repeated on four clusters it is possible to only observe a difference in the minimum CLI allowed for the most conservative threshold on ZU9EG, which is reduced to 0.2%. In all the other cases the CLI values

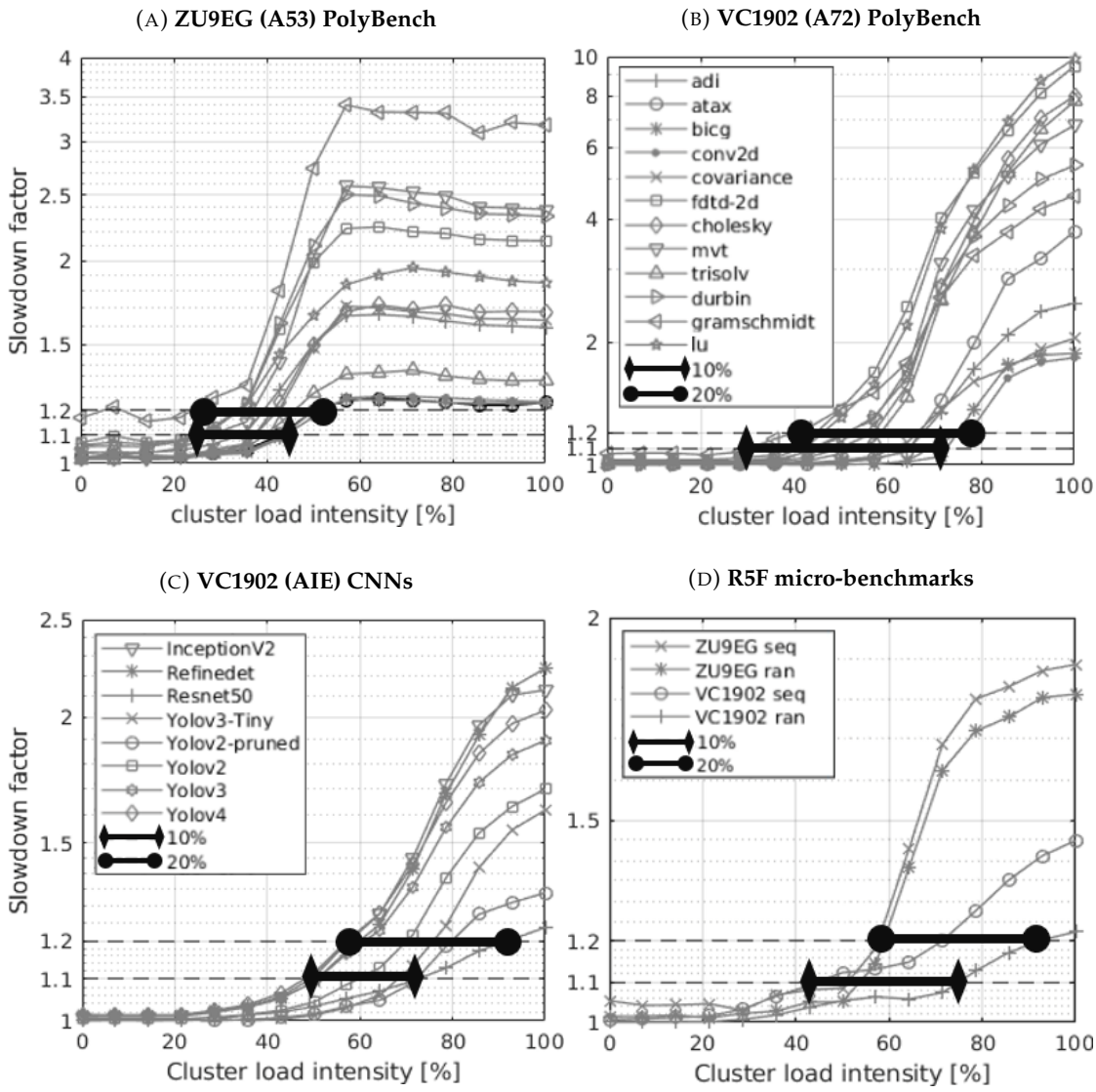


FIGURE 5.5: *CMRI* on different Processing Elements and platforms. (5.5a and 5.5b): Polybench on APUs; (5.5c): CNNs on AI Engines; (5.5d): Micro-benchmarks on R5F.

remain nearly identical, but the maximum bandwidth to which they refer is in this case much higher (see Figure 4.10): 14 GB/s for ZU9EG and 24 GB/s for VC1902.

On the RPUs (Figure 5.5d) CMRI allows to exploit 41-to-78% of the maximum bandwidth for a single cluster on both SoCs for the 10% threshold, and 59-to-93% for the 20% threshold. If four clusters are considered, CMRI allows to exploit 8-to-49% of their maximum bandwidth for the 10% threshold, and 29-to-54% for the 20% threshold.

On the AIE (Figure 5.5c) CMRI allows to exploit 50-to-72% of the maximum bandwidth that a single cluster can generate on VC1902 for the 10% threshold, and 59-to-93% for the 20% threshold. Even when considering four active clusters the CLI

values remain nearly identical, which means there are cases where, nearly the whole bandwidth generated by the FPGA can be exploited, without significantly impacting the timing of the CNN workloads.

Improvements on bandwidth exploitation

After having experimentally shown the capability of the presented methodology to mitigate memory interference, the experiments presented below are more focused in highlighting the benefit in terms of memory bandwidth exploitation. Here the reference is the PREM approach [55], in which only one task at a time is allowed to access main memory. For this reason, even a small increase in bandwidth usage is an improvement respect to the baseline. Also in this case the experiments have been carried out on the two reference platforms, the Zynq UltraScale+ MPSoC and Versal ACAP.

The plots shown in Figures 5.6, 5.7, 5.8 and 5.9 combine information about the bandwidth requested by the FPGA accelerator clusters (shown as coloured areas) and the latency of the task under test (shown as curves). The x-axis of these plots shows a variation on the CLI. The plots show two coloured areas, the red area represents the case where only one accelerator cluster is active during the test. Cavicchioli et al. [16] observe that the most convenient way of implementing PREM in a GPU-based HeSoC like the *NVIDIA TX2* is that of allowing one core from each of two CPU islands to operate at the same time, as they don't interfere much. Following this scheme, the minimum that can be done in absence of CMRI mechanisms is that of allowing a single FPGA accelerator to operate in parallel with a single CPU core. For the considered platforms this scheduling scheme is already too coarse in terms of the generated bandwidth. The blue area represents the case where all four clusters are active during the experiment, that has been included to study where there ever is a case in which the FPGA might safely run at full throttle in parallel with the CPU cores. The reference bandwidth values for the coloured areas is to be read on the y-axis on the left. The plots also present two coloured curves, which show the slowdown suffered from the task under test running on one CPU core when one (red curve) or four FPGA accelerator clusters (blue curve) are considered. The slowdown value is shown on the y-axis on the right. It is important to note that: (i) this axis has a logarithmic scale; (ii) both y axes are stretched vertically beyond a certain value (6 GB/s for the one on the left, 2× slowdown for the one on the right). The plots have

been stretched to emphasize what happens in the lower part of the graphs, which is the most important.

To analyse if, given these two thresholds, there exist working points for the accelerator clusters to use DRAM bandwidth, the focus is on the point where the horizontal lines intersect the slowdown curve. The intersection point shows on the x-axis the maximum throttle (CLI) that can be applied to the FPGA accelerators. From there, a vertical line has been projected up to the top border of the corresponding coloured area. The tip of the arrow indicates the maximum bandwidth that the FPGA cluster(s) can actually use.

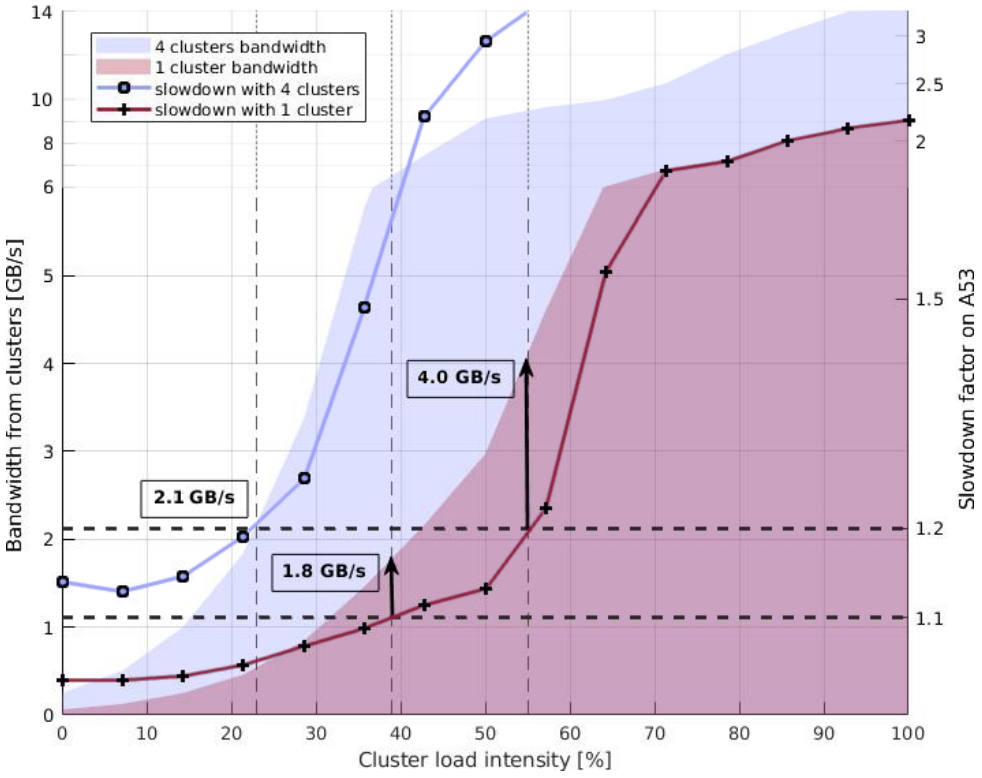
Considering the Zynq UltraScale+ platform shown in Figure 5.6 and 5.7, by using the CMRI technique, FPGA clusters that exploit up to 4.5GB/s and 5.1GB/s can be co-scheduled with respectively *sequential* and *random* under test applications. The new Versal ACAP platform, shown in Figure 5.8 and 5.9, on the other hand, showed more room to improve memory bandwidth exploitation than the previous generation of MPSoC. In this case, the FPGA clusters can use up to 4.3GB/s and 8.6GB/s for the *sequential* and *random* cases.

These experiments showed that CMRI, previously only applied to multi-core CPUs, is very effective also in FPGA-based HeSoCs, allowing in the best case to exploit up to 8.6GB/s of the available bandwidth to the FPGA accelerators, while providing analogous timing guarantees to what is offered by PREM-like schemes [55, 14]. The presented results, also highlight that the proposed methodology is more effective for applications that exhibit a pseudo-random memory access pattern, as it is less sensitive to interference.

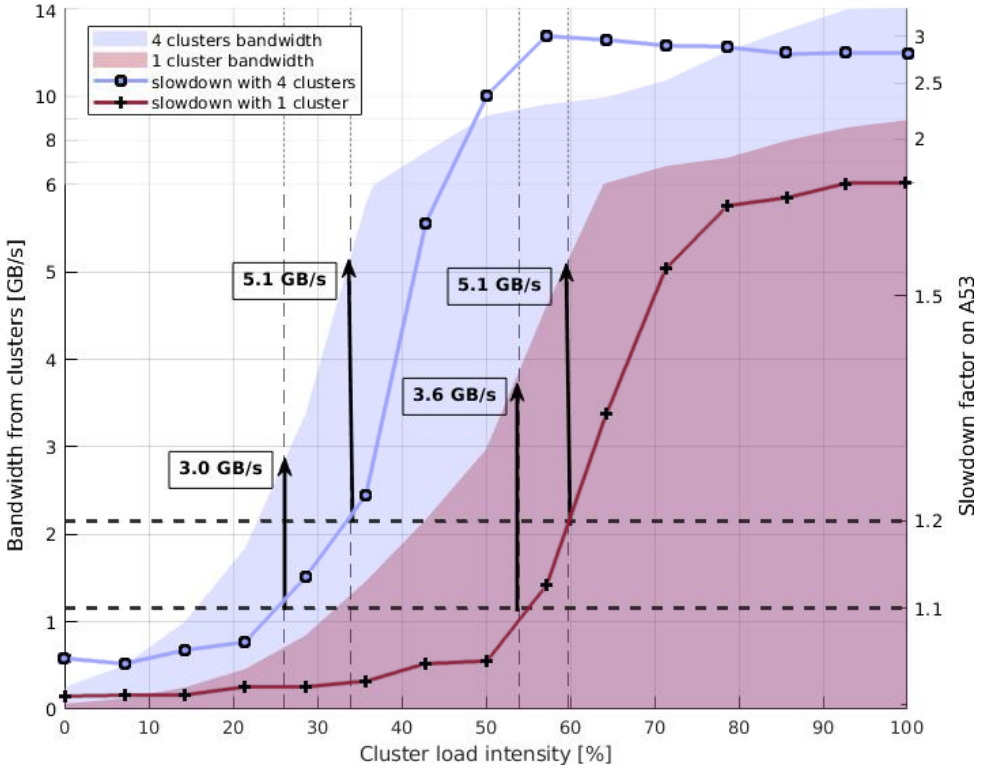
5.3.2 Cost of HW and SW Monitoring and Throttling

Implementing monitoring in software can be done by relying on the *Xilinx AXI Performance Monitor (APM)*[5] a commercial solution integrated on Zynq UltraScale+ devices. The latter can be read in active polling mode on a RPU core (to avoid any performance impact on the applications, and because RPUs offer in general more predictable behavior than the APUs). Figure 5.10 shows the hardware design used for this comparison. The sniffer has been replaced with an APM. The RPU core can interact with the Proxy-Core through the *On-Chip-Memory (OCM)*, to receive the configuration parameters. Reading from/writing to a APM register takes 0.32us on the

[5] https://www.xilinx.com/support/documentation/ip_documentation/axi_perf_mon/v5_0/pg037_axi_perf_mon.pdf

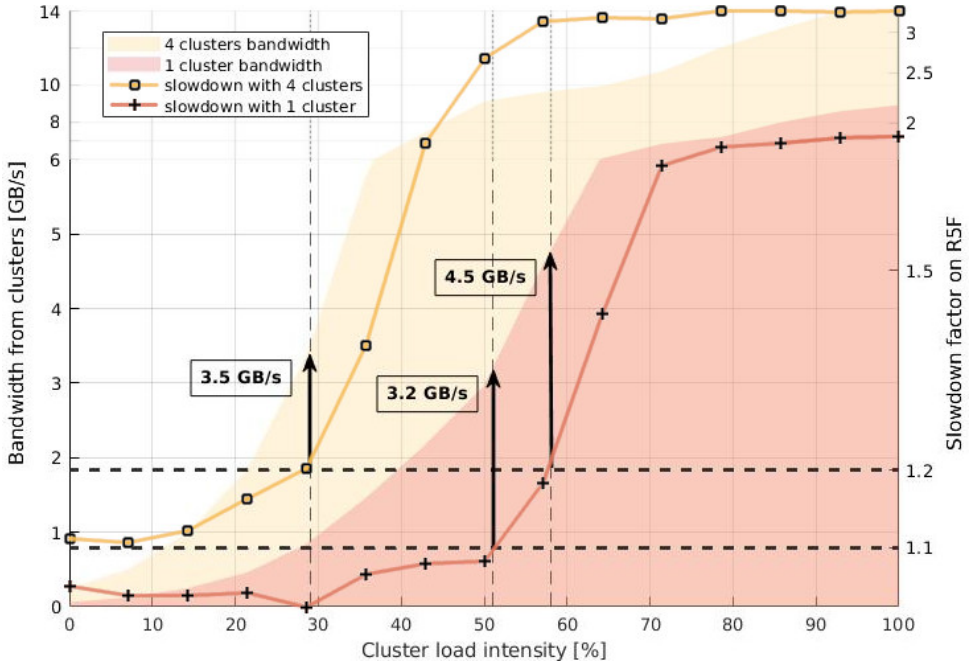


(A) **UT: *sequential* access pattern**

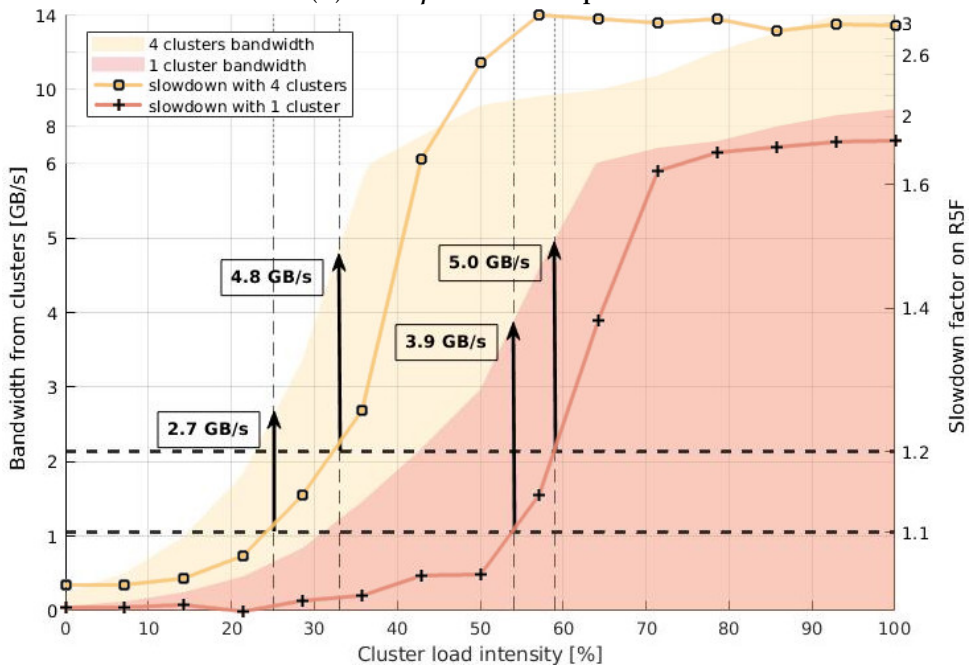


(B) **UT: *random* access pattern**

FIGURE 5.6: Analysis of the memory bandwidth exploitation improvements compared to PREM-like schemes. The tasks under test run on *ARM Cortex A53* of the *Zynq UltraScale+ MPSoC*.

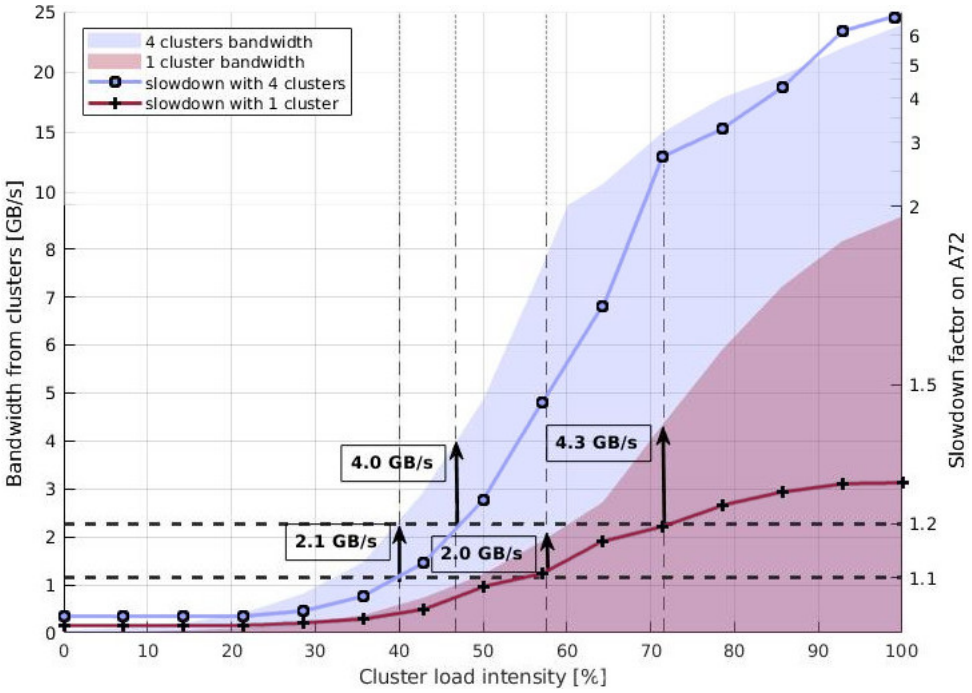


(A) **UT: *sequential* access pattern**

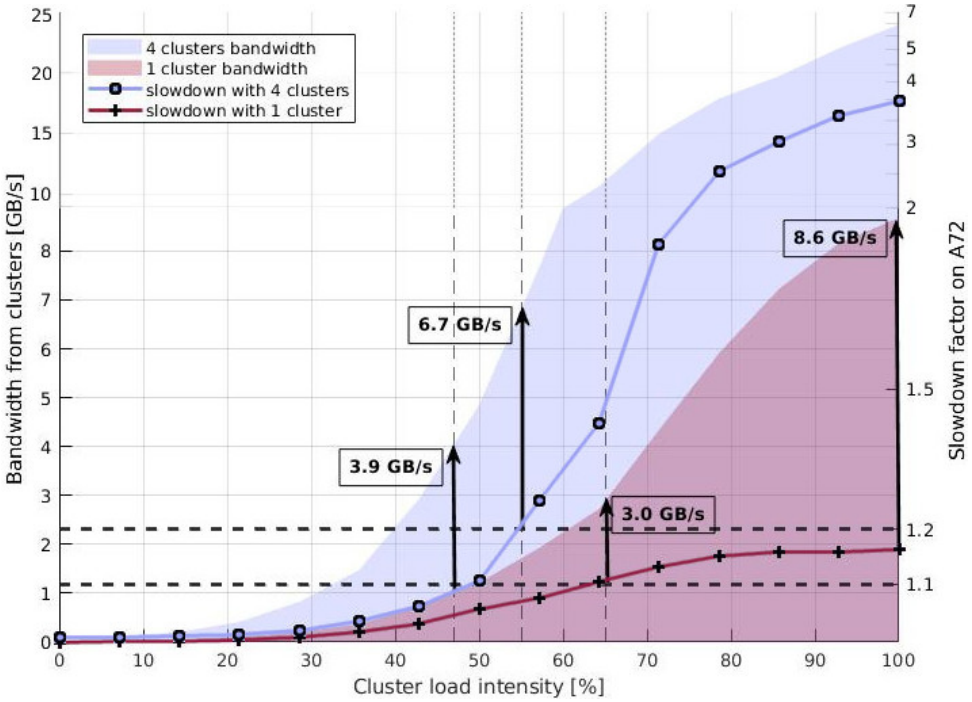


(B) **UT: *random* access pattern**

FIGURE 5.7: Analysis of the memory bandwidth exploitation improvements compared to PREM-like schemes. The tasks under test run on *ARM Cortex R5F* of the *Zynq UltraScale+ MPSoC*.

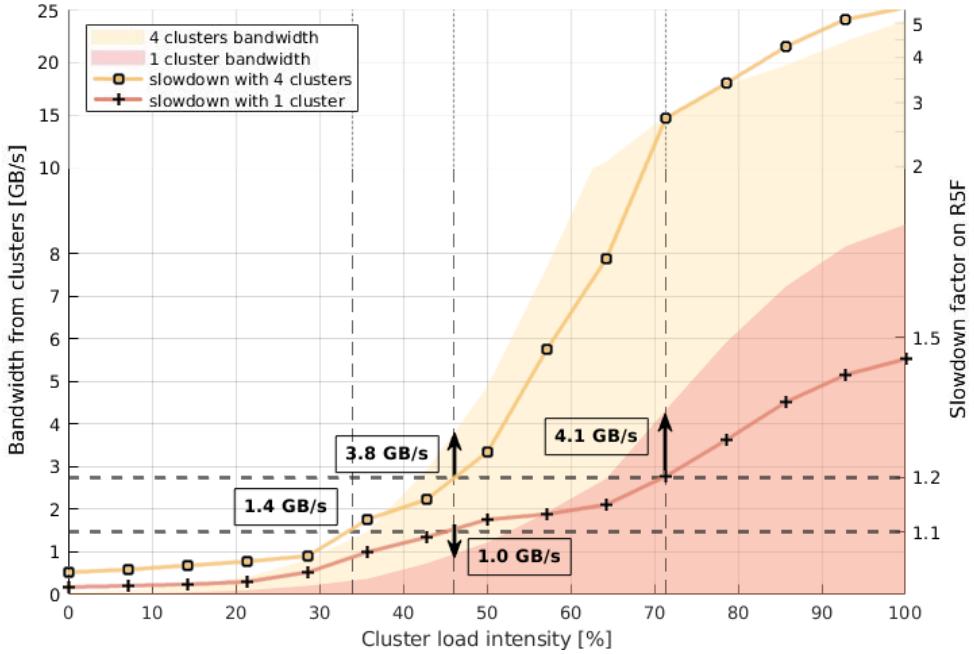


(A) **UT: *sequential* access pattern**

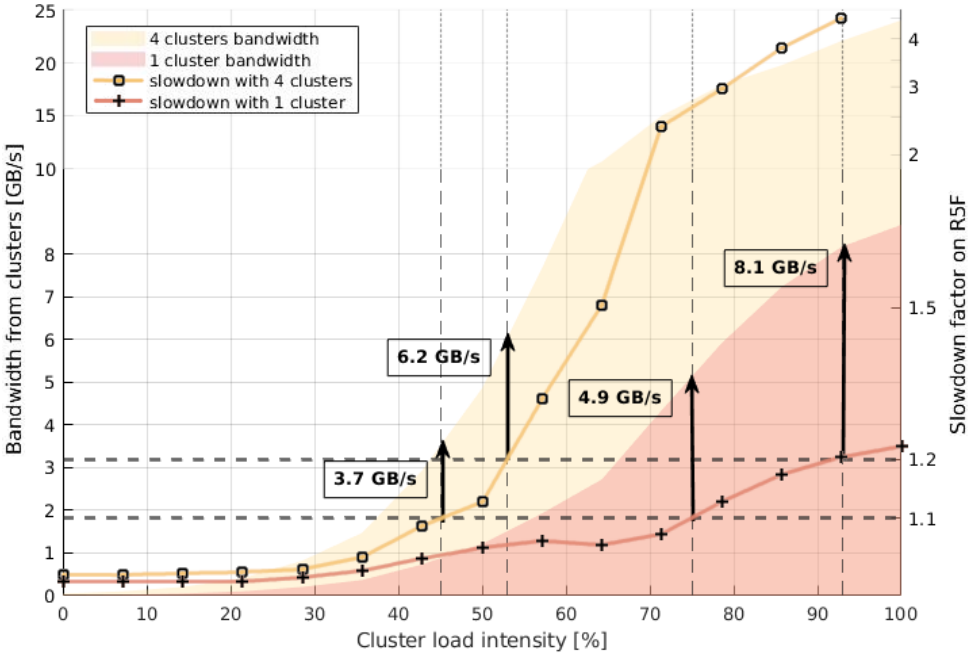


(B) **UT: *random* access pattern**

FIGURE 5.8: Analysis of the memory bandwidth exploitation improvements compared to PREM-like schemes. The tasks under test run on *ARM Cortex A72* of the *Versal ACAP*.



(A) **UT: *sequential* access pattern**



(B) **UT: *random* access pattern**

FIGURE 5.9: Analysis of the memory bandwidth exploitation improvements compared to PREM-like schemes. The tasks under test run on *ARM Cortex R5F* of the *Versal ACAP*.

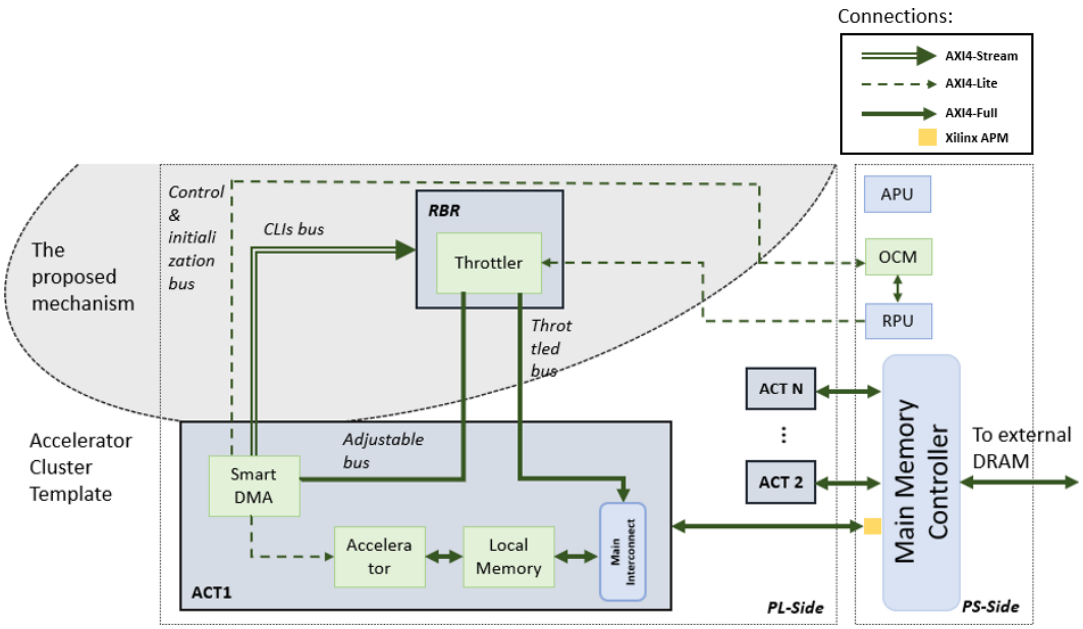


FIGURE 5.10: ACT based on AXI Performance Monitor (APM) used to compare with our methodology.

RPUs. During this time our DMA design can transfer 512B, which has been used as a minimum transfer size. It always takes at least two iterations (i.e., two APM register reads) for the monitoring system to detect a transfer size of 512B, as it does not operate in full sync with the DMA. A third read operation follows to a different APM register to retrieve the exact time elapsed since the beginning of the transfer. The software routine can then compute the amount of idleness required. Implementing throttling in software can be done by partitioning a single DMA transfer in several smaller ones interspersed with idle cycles, as highlighted in the previous chapter (Listing 4.3). Figure 5.11 shows the performance penalty to implement such scheme as DMA transfers get smaller and more numerous (idleness is 0% here). Transferring 512KB in 1024 chunks of 512B each costs ten times a single transfer of 512KB, requiring around 3ms. As the throttling scheme relies on the active time to determine the idle time, if the worst-case is considered, 1% CLI requests the SW throttling approach would become ready to process a new request after $3 * 100 = 300ms$, which is too coarse for a RTOS. The HW throttler proposed in this chapter does not imply overheads for partitioned DMA transfers, and effectively uses the time to transfer 512B (0,32us) as a baseline for the idle time calculation. Considering the worst-case 1% CLI request the HW throttling approach would become ready to process a new request after $0,32 * 100 = 32us$. This is in line with the most stringent RTOS scheduler tick in use today.

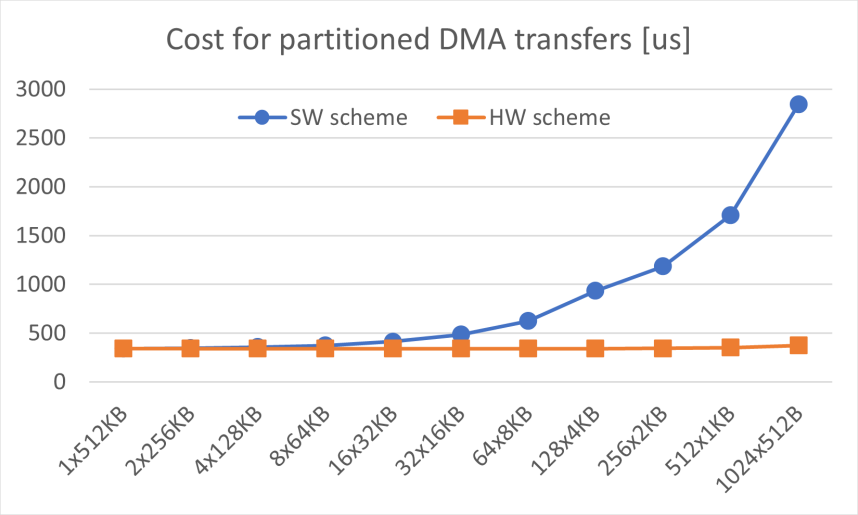


FIGURE 5.11: SW and HW Throttling Costs (in *us*) to partition a DMA transfer in multiple smaller transfers.

5.3.3 Tightly-coupled versus Loosely-coupled monitoring and throttling

To test the impact on performance and granularity of the tightly-coupled operation of the *monitor* and the *throttler*, a comparison has been proposed. The aim is to test the integrated solution (TCMT) with one where the HW *throttler* is coupled to a SW *monitor* that relies on the APM (LCMT). Listing 5.5 shows the C code of the control software running on the RPU core for the LCMT scheme.

```
1   XAxiPmon_SetMetrics(&AxiPmonInst, PORT_S3, XAPM_METRIC_SET_2
        , XAPM_METRIC_COUNTER_1);    // S3 - Write Byte Count
2   XAxiPmon_SetMetrics(&AxiPmonInst, PORT_S3, XAPM_METRIC_SET_6
        , XAPM_METRIC_COUNTER_4);    // S3 - Total Write Latency
3
4   XAxiPmon_ResetMetricCounter    (&AxiPmonInst);
5   XAxiPmon_ResetGlobalClkCounter(&AxiPmonInst);
6
7   XAxiPmon_StartCounters(&AxiPmonInst, SAMPLE_INTERVAL);
8
9   u32 copy_cycles_s = 0;
10  u32 copy_cycles_e = 0;
11
12  for (int i = 0; i < MAX_RUNS; i++) {
13
14      u32 new_threshold = (i+1)*cpy_threshold;
15
```

```
16          while(XAxiPmon_GetMetricCounter(&AxiPmonInst,
                XAPM_METRIC_COUNTER_1) < new_threshold) { /* waiting
                 */ }
17
18          copy_cycles_e = XAxiPmon_GetMetricCounter(&AxiPmonInst,
                XAPM_METRIC_COUNTER_4) - copy_cycles_s;
19
20          // write copy_cycles
21          Xil_Out32(GPIO_BASEADDR + 1*XGPIO_CHAN_OFFSET +
                XGPIO_DATA_OFFSET, copy_cycles_e);
22          // raise IDLE_valid
23          Xil_Out32(GPIO_BASEADDR + 0*XGPIO_CHAN_OFFSET +
                XGPIO_DATA_OFFSET, 1);
24          Xil_Out32(GPIO_BASEADDR + 0*XGPIO_CHAN_OFFSET +
                XGPIO_DATA_OFFSET, 0);
25
26          copy_cycles_s += copy_cycles_e;
27      }
28
29      XAxiPmon_StopCounters(&AxiPmonInst);
30  }
```

LISTING 5.5: Software routine to interact with APM registers

Lines 1-10 show the initialization part of the APM peripheral, which is performed only once. The `for` loop between lines 12-27 implements the actual memory bandwidth monitoring routine, whose logical operation is the same as implemented in FPGA inside the *RBR*. On line 16 a waiting `while` loop is implemented, which stops the execution of the software until a configurable threshold (in Bytes) is reached. Lines 18-21 allow respectively the reading of the time needed (in clock cycles) for the memory transfer of $N-$bytes and the subsequent transfer of the information towards the ACT design. The latter operation involves a register write. Finally Lines 23-24 implement the `idle_valid` signal notification to identify the end of a memory chunk transfer.

Figure 5.12 shows the outcome of the test. The black dashed curve represents an hypothetical trace of CMRI values imposed by the system scheduler over time. To stress the worst case, the assumption is that the trace evolves with the same minimum frequency at which the integrated HW mechanism handles the worst-case 1% CMRI request (i.e., every 32us, see previous subsection). The progress of time

TABLE 5.2: Overheads of the software monitoring routine presented in Listing 5.5

C Code (Lines)	Assembly	Clock Cycles (100MHz)	Time (μs)
14-16	*add x0, sp, #0x48* *mov w1, #0x1* *bl 30a8 <XAxiPmon_GetMetricCounter>* *cmp w0, w21* *b.cc 16e70 <get_bytes_start>*	68	0.68
18	*mov w1, #0x4* *add x0, sp, #0x48* *bl 30a8 <XAxiPmon_GetMetricCounter>*	37	0.37
21	*mov x1, #0x8* *movk x1, #0xa000, lsl #16* *str w0, [x1]*	32	0.32
23-24	*mov x0, #0xa0000000* *mov w1, #0x1* *str w1, [x0]* *str wzr, [x0]*	61	0.61

is shown on the X axis, while the Y axis shows the requested CMRI (black curve) and the measured bandwidth in GB/s for (i) TCMT (blue, star-shaped markers); (ii) LCMT (red, square-shaped markers); (iii) LCMT slow (green, asterisk-shaped markers). As the experiment and the plot have been constructed based on the worst-case 1% CMRI request, the various schemes are expected to struggle the most at low CMRI rates, as the timing is such that a single request can be processed before the system evolves. The higher the CMRI becomes, the higher the number of times each scheme can be queried within the same system state (i.e., within the same CMRI request). To investigate this intuition, the graph considers three distinct areas: (i) bottom left: the CLI ranges between 1% and 10%; (ii) center: the CLI ranges between 30% and 50%; (iii) top high: the CLI ranges between 50% and 100%.

TCMT is in general able to precisely follow the CLI profile in every operating condition. Intuitively, there might be some errors in the lower CLI ranges, as the timing of the system scheduler that imposes the CLI values might not be fully aligned with that used internally by TCMT. This justifies the small undershoot visible in the CLI range 1% - 10%, a close-up of which is visible in the sublot in the top left area of Figure 5.12. Here, the misalignment between the said timers causes TCMT to sample the CLI value just before the system state changes. Since the CLI value is very low $\approx$

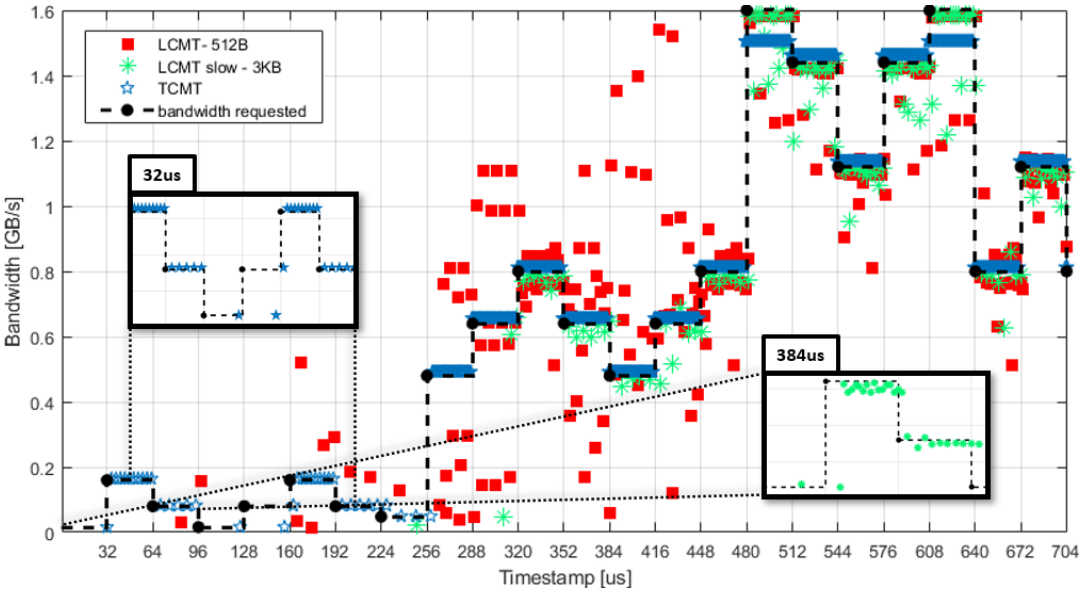


FIGURE 5.12: Comparison of our tightly-coupled solution with loosely-coupled SW mechanism, to follow a temporized bandwidth profile.

1%, TCMT will cause the DMA to stay idle for the longest time, which justifies the delay in following the CLI profile.

Note that it is sufficient to slow down the scheduler frequency to one sample every 64 microseconds to solve the problem, while still matching the requirements of the most stringent RTOS scheduler.

The red samples show what happens if the same scheduler tick of 32us is applied to LCMT. As explained from the previous subsection (and as can be observed from Table 5.2), the interaction between the RPU control software, the APM registers and our HW throttler has a longer execution cycle compared to our integrated TCMT solution, which makes the system less responsive and prone to errors. The fact that the system is less responsive can be easily observed in the area to the bottom left. Here the number of red samples is much smaller than that of the blue samples. This is because LCMT takes a longer time to complete the check on the 512B transfer threshold and to then compute the idle cycles and initiate the throttling. The code reported in Listing 5.5 shows that the SW requires six between reads and writes, which take six times the latency of TCMT. This also justifies the fact that the system is prone to errors, which is easily observed by the very wide and numerous over/undershoots throughout the CLI profile. As LCMT takes a longer time to complete the check on the 512B transfer threshold, when it finally realizes the threshold has been reached the system has in fact copied a significantly larger amount of data (this justifies the overshoots). This implies that at the following iteration the threshold will already

be much closer, possibly requiring little to no copy activity (this justifies the undershoots). Note that this poor degree of control offered by LCMT can lead to deadline misses, ultimately making the approach unusable in timing critical systems.

To make LCMT reliable the system scheduler tick must be slowed down such that it can accommodate a complete worst-case BW monitoring + throttling cycle (CLI=1%). According to our previous observation, it would be sufficient to check the copy transfer threshold six times slower for the LCMT technique to become stable. The green samples shows what happens when this threshold is set to 3KB (i.e., $6 \times 512B$). While these numbers confirm that the intuition is correct, they also suggest that LCMT requires further slowing down of the scheduler tick to fully stabilize. We have experimentally measured that the system scheduler tick must be slowed down to one sample every 384 microseconds for the LCMT technique to reliably follow the bandwidth profile: 12$\times$ slower than the TCMT technique. Although this is still within the same order of magnitude of what the most stringent RTOS scheduler require, it must be noted that our experiments don't take into account additional OS-level overheads for the CLI decisions to be taken and propagated to our monitoring and throttling components. The sublot in the bottom righ area of Figure 5.12 shows a close-up of the most critical CLI request range 1%–10% depicting LCMT slowed down by a factor of 12$\times$ compared to TCMT.

5.3.4 QoS for memory interference mitigation

The third experiment involves the use of the fine-grained bandwidth regulation for the mitigation of memory interference and to provide QoS guarantees when co-scheduling HW and SW tasks in the whole platform. To this aim, are presented five scenarios corresponding to as many QoS levels for the various tasks, each tolerating up to X% performance degradation (slowdown): (i) *Very-Tight (VT)*, where X=20%; (ii) *Tight (T)*, where X=40%; (iii) *Moderate (M)*, where X=60%; (iv) *Loose (L)*, where X=80%; (v) *Very-Loose (VL)*, where X=99%. The system configuration is composed of three *Accelerator Clusters (ACTs)*, one for each port of the DRAM accessible from the FPGA. The accelerators are configured to perform only memory reads and the design is synthesized with a frequency of *300MHz* and each ACTs has an RBR connected.

TABLE 5.3: Real-world benchmarks

Scenario	FPGA		APU		RPU	
	ACT1	**ACT2**	**MM**	**MT**	**VMA**	**I2C**
VT	1.20×	1.20×	1.22×	1.07×	1.70×	1.11×
	1.20×	-	1.22×	1.07×	1.22×	1.02×
	-	-	1.22×	1.07×	1.12×	1.01×
T	1.38×	1.38×	1.22×	1.07×	1.35×	1.04×
M	1.59×	1.59×	1.22×	1.07×	1.31×	1.04×

Comparison with Zynq UltraScale+ QoS ecosystem

From the experimental setup point of view the configuration proposed by A. Serrano-Cases et.al [67] was adopted. Besides the ACTs there are also different workloads on the host cores in the system, in particular: two cores of the ARM Cortex A53 that perform a matrix multiplication (MM) and a matrix transpose (MT) respectively. On the other side two cores of the ARM Cortex R5, perform a vector add (VMA) and an image to column (I2C) operation. Table 5.3 shows an analysis of the memory interference on the experimental setup described above. The table shows a slowdown value for each processing element that needs to remain within the considered thresholds. ACT3 is considered best-effort (BE), therefore it is not included in the table. In this case ACT3 has been configured to perform memory reads at 5% of its possibility (CLI = 5%). The first row considered in the table refers to the 80% performance guaranteed for all tasks (apart from ACT3 which is BE). In this case it is not possible to provide guarantees to all the processing elements involved. Even relaxing the constraints on ACT1 and ACT2, making them BE it is not possible to provide guarantees on the execution time of the other tasks, due to intra-core interference. For the other two scenarios considered (T and M), the slowdown of the tasks deployed on the system always remains within the constraints of 40% and 60%. This represents a significant improvement over bandwidth regulation techniques only based on QoS mechanisms offered by the Zynq UltraScale+ platform [67].

Real-world benchmarks

The capability to mitigate memory interference were also evaluated in more realistic contexts where 8 computational kernels of the *PolyBench* tool suite have been considered. In this experiment the kernels are grouped into two different tasksets, called: `LOW` and `HIGH` with respect to the sensitivity of the benchmark to memory interference. Table 5.4 shows the two different groups of tasks, where given a group, each of

benchmark	maximum slowdown	taskset
atax	1.4×	LOW
trisolv	1.7×	
bicg	1.3×	
conv2d	1.5×	
adi	2.0×	HIGH
cholesky	2.3×	
covariance	1.9×	
fdtd-2d	3.1×	

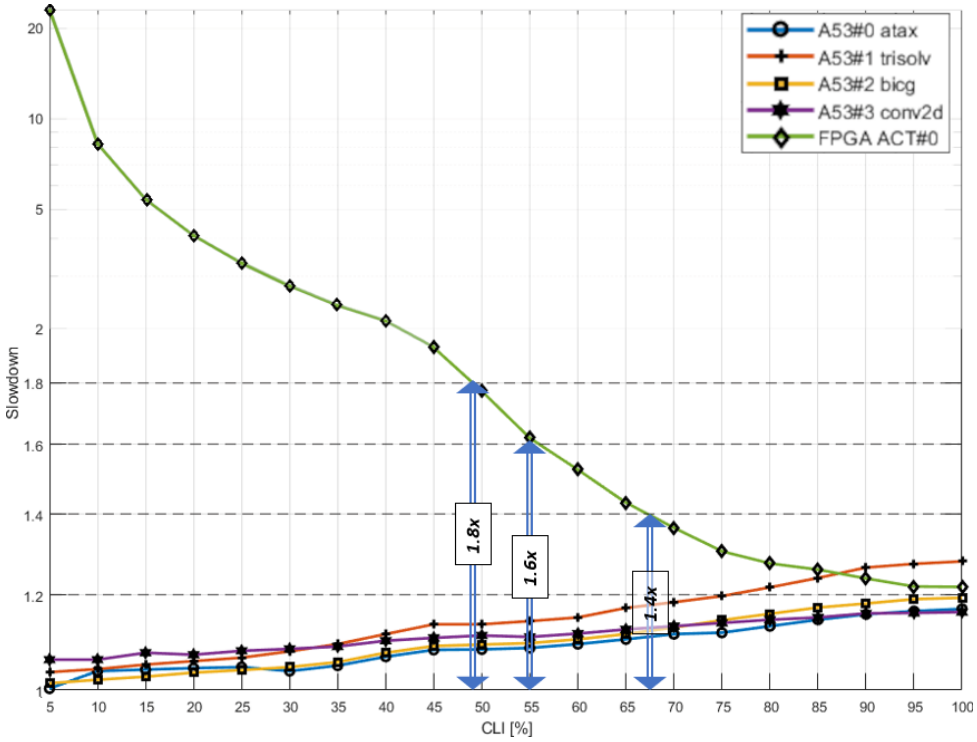
TABLE 5.4: PolyBench kernels grouped on two different taskset, based on interference sensitivity.

the four task inside this group is mapped on the four ARM Cortex A53 of the Zynq UltraScale+. For example the `LOW` category contains the four benchmarks that exhibit the lowest slowdown due to interference. The experiments are shown in Figure 5.13 and 5.14. For each experiment two graphs are presented, which respectively show a scenario where only one ACT is active (subfigure *A*), or two ACTs active at the same time (subfigure *B*). The ACTs are synthesized with *300MHz* frequency and in this case each ACT is configured to perform concurrent readings and writings in main memory.

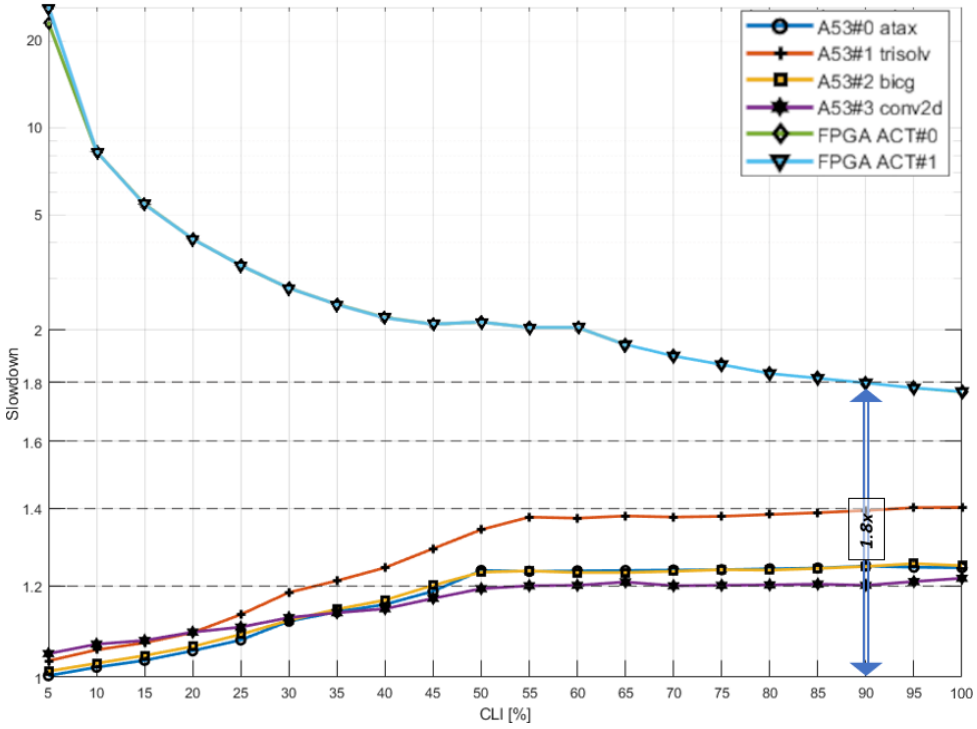
For each graph, the regulation value (CLI) is shown on the X axis, considering the range $0.05 \leq CLI \leq 1$. The slowdown of the tasks is shown on the Y axis, where four thresholds have been identified *VT*, *T*, *M*, *L* considering the scenarios previously introduced. The experiments show that for the `LOW` taskset, shown in Figure 5.13, there is at least one feasible configuration in each of the two considered scenarios, highlighted by the labeled arrows on the graphs. In the `HIGH` taskset, on the other hand, due to the higher sensitivity to interference, there is only one possible solution, in the case where only one ACT is scheduled in the system (Figure 5.14a) for the *L* configuration, which corresponds to 80% of maximum tolerated interference. The `HIGH` taskset has been omitted, since due to the high sensitivity to memory interference, was not possible to control the CPU task slowdown, even in the case where all the ACTs are off.

Synthetic benchmarks

Table 5.5 shows the interference measured on the same scenarios considered above, but in this case the same benchmark is used on all cores. Specifically, it is considered a synthetic benchmark, where all tasks perform memory reads with configurable

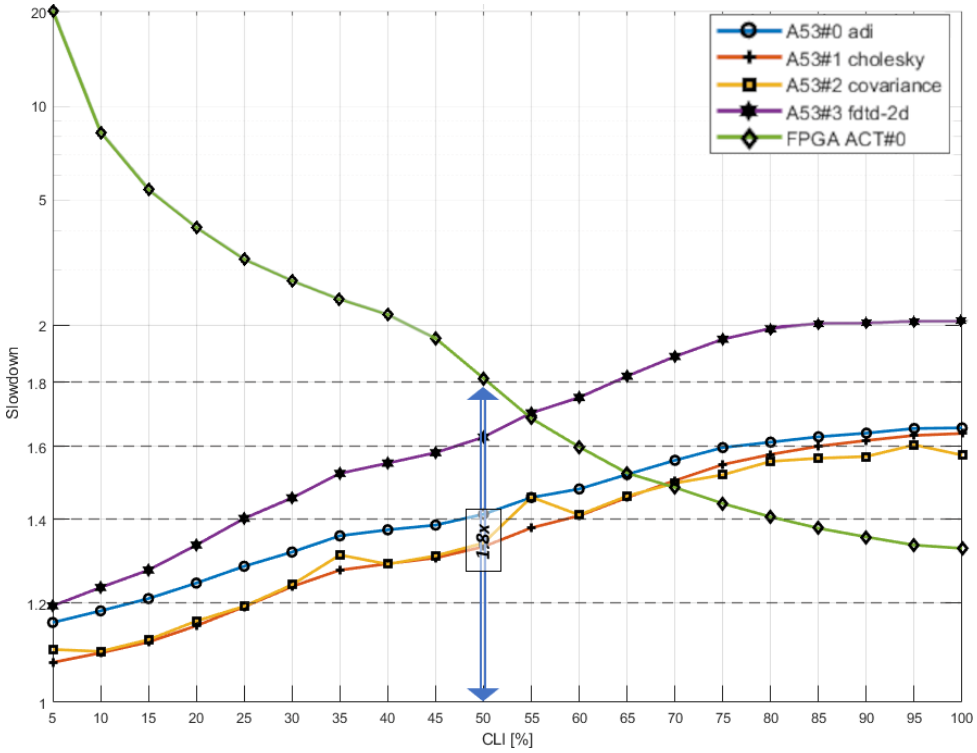


(A) Only one ACT deployed on the system.

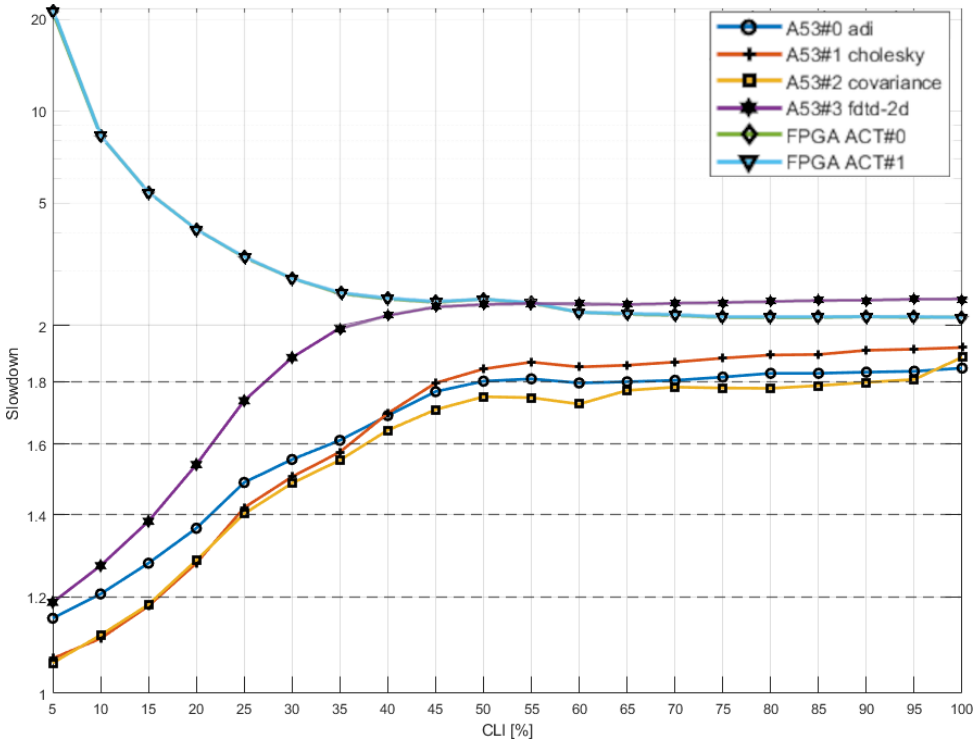


(B) two concurrent ACTs deployed on the system.

FIGURE 5.13: Co-scheduling of LOW sensitivity tasks when running 1 and 2 concurrent ACTs.



(A) Only one ACT deployed on the system.



(B) two concurrent ACTs deployed on the system.

FIGURE 5.14: Co-scheduling of HIGH sensitivity tasks when running 1 and 2 concurrent ACTs.

TABLE 5.5: Synthetic memory-intensive workloads

Scenario	FPGA		100% memory-intensive APU				100% memory-intensive RPU				75% memory-intensive APU				75% memory-intensive RPU			
	ACT1	ACT2	A53#0	A53#1	R5#0	R5#1	A53#0	A53#1	R5#0	R5#1	A53#0	A53#1	R5#0	R5#1	A53#0	A53#1	R5#0	R5#1
VT	1.20×	1.20×	1.63×	1.63×	1.67×	1.67×	1.54×	1.52×	1.52×	1.52×	1.01×	1.04×	1.08×	1.07×				
	1.20×	-	1.37×	1.37×	1.35×	1.35×	1.28×	1.29×	1.28×	1.28×	1.01×	1.02×	1.05×	1.03×				
	-	-	1.21×	1.22×	1.22×	1.22×	1.2×	1.2×	1.17×	1.18×	1×	1×	1×	1×				
T	1.40×	1.40×	1.51×	1.52×	1.51×	1.51×	1.43×	1.41×	1.39×	1.4×	1.04×	1.06×	1.07×	1.06×				
	1.39×	-	1.33×	1.32×	1.33×	1.33×	1.29×	1.28×	1.26×	1.25×	1.01×	1.03×	1.02×	1.03×				
	-	-	1.21×	1.22×	1.22×	1.22×	1.2×	1.2×	1.17×	1.18×	1×	1×	1×	1×				
M	1.60×	1.60×	1.46×	1.44×	1.51×	1.51×	1.4×	1.4×	1.38×	1.38×	1.02×	1.02×	1.01×	1.04×				
	1.60×	-	1.31×	1.32×	1.34×	1.34×	1.3×	1.32×	1.26×	1.27×	1×	1×	1×	1.01×				
	-	-	1.21×	1.22×	1.22×	1.22×	1.2×	1.2×	1.17×	1.18×	1×	1×	1×	1×				

computation to communication ratio (x% memory-intensive). Where memory intensity is 100% the benchmarks perform memory reads at their full throttle. In the other two cases, 25% and 75% of computation is inserted. The benchmarks are tuned to maximize the number of cache misses, to ensure the issued requests are in fact serviced from the DRAM (and not intercepted by the cache hierarchy). The memory reads are performed with stride equal to the L2-Cache line size. In this case considering the VT scenario, for 100% and 75% of memory-intensity, it is not possible to guarantee 80% performance for all system tasks, even considering all three ACTs as best-effort. Regarding the T scenario, it becomes feasible only when two out of three ACTs are considered BE. On the other hand, all the other scenarios considered (M, L and VL) are feasible even in the most constrained case, where only ACT3 is BE. Finally the least memory intensive (25%) is feasible in all considered scenarios.

5.4 Conclusion

This chapter introduced a fine-grained QoS control via tightly-coupled bandwidth monitoring and regulation for FPGA-based HeSoCs. The innovative mechanism is based on an original IP design, called Runtime Bandwidth Regulator. Comparerd to SW-only bandwidth regulation mechanisms our design is 2 orders of magnitude faster, reliably adapting to dynamically varying QoS level requests with much higher frequency. Even when compared to more loosely-coupled, state-of-the-art badnwidth regulation mechanisms on the Xilinx Zynq UltraScale+ HeSoCs, the proposed RBR mechanism can operate up to 12 times faster. The proposed mechanism was able to accurately control the bandwidth of the accelerator cluster templates and to guarantee the required QoS level for the tasks running on FPGA under every considered system load scenario. Moreover, our experiments have shown that, even if it cannot directly throttle the bandwidth of the SW-tasks, the proposed mechanism has

a controllable, indirect effect also on these. Finally, was reported how the proposed mechanism is capable of guaranteeing QoS requests that were deemed as not feasible with bandwidth regulation mechanism currently available on Zynq Ultrascale+ [67].

Chapter 6

Real-World Embedded Applications

This chapter, more application-oriented compared to the others discussed in this thesis, presents two applications in the embedded domain. The two use-cases considered are evaluated using state of the art heterogeneous systems, and are designed employing memory intensive accelerators, which if not properly regulated would lead to interference. The first application concerns the localization of an autonomous robot within an environment. This application is an example of a highly parallel and memory-intensive workload that is a perfect candidate to be implemented on a dedicated accelerator. To implement the robot localization an FPGA-based accelerator, integrated within a cluster, will be presented. The cluster is organized following the Proxy-Core based ACT presented in chapter 4. The engine is controlled by a *Xilinx MicroBlaze* that has been used as a Proxy-Core. The Proxy-Core can exploit local data fetched by a dedicated DMA engine, into BRAM banks.

The second application considered involves Convolutional Neural Networks to solve the object detection and classification tasks. This second use case represents an application of Artificial Intelligence (AI), typically used in the field of CPS like self-driving cars, autonomous UAVs, Industry 4.0 and many others. CNNs are by nature complex models that typically require dedicated accelerators (e.g. GPGPUs, FPGAs or ASICs). Dedicated accelerators can be considered as memory bound engines[1] (like DMAs) that, as already explained in chapter 4, if not properly regulated, they could be affected by unbounded slowdowns due to contention. Neural accelerators are typically organized following a clustered architectural design. The ACT structure presented in chapter 4 and the RBR built on top of that are general enough

[1] well designed accelerators make use of double buffering techniques, such that memory transfers are continuously issued and overlapped with computation.

to be applied also in this context.

6.1 Particle Filter localization

Different from software-programmed GPUs, the main limiting factor in the adoption of FPGAs is the complex development process. This workflow can be further complicated in the context of HeSoCs, due to complex interactions between the system components. *High Level Synthesis (HLS)* is increasingly being adopted to ease IP design, allowing engineers to quickly prototype their solutions. However, automated tools still lack the capability to build a complete hardware-software enviroment. One alternative to overcome these limitations and improving design productivity is to use a virtual hardware representation that overlays the original FPGA fabric, referred to as an *overlay* architecture [33]. Overlays are programmable, coarse-grained hardware abstraction layers on top of the FPGA hardware, abstracting the underlying hardware details as a software-managed task, which can be hooked to standard APIs for heterogeneous compute platform programming. **The Accelerator Cluster Template (ACT) discussed in section 4.3.1 is an example of these overlays**, where the configuration of the accelerators and memory transfers are orchestrated by means of the Proxy-Core. This application can be considered as an use case of what presented in the previous chapters. The technology and ideas proposed in this chapter are the result of a teamwork, carried out within the *High Performance Real-Time Laboratory (HiPeRTLab)*. The accelerator presented in this section is currently adopted by the HiPeRT team in the *F1/10th*[2] international competition.

6.1.1 FPGA-Overlay for Real-Time Localization

This section discusses how the previously described ACT, can be adapted and tailored to deploy a fully functional AV stack in a simple and effective, but most important *scalable* manner, targeting a state-of-the-art heterogeneous FPGA-based SoC, the Xilinx Zynq UltraScale+ (see chapter 2), as the *Domain Controller* of a 1/10th scale racing vehicle prototype [17, 50]. We provide a solution for offloading the *localization* component of a typical AV driving stack, where the vehicle matches the data coming from perception modules (i.e., from a LiDAR sensor) with a pre-built map for precise localization. We employ a well-known Monte-Carlo method called Particle

[2] https://f1tenth.org/

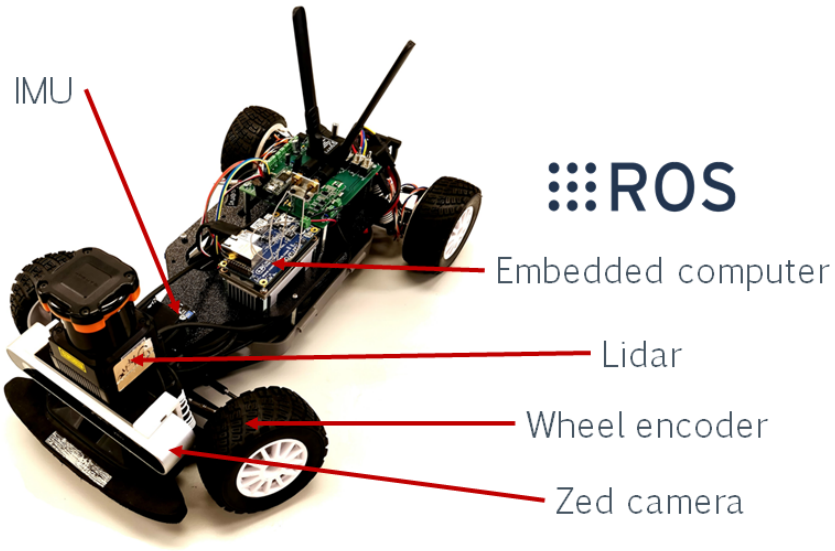


FIGURE 6.1: The F1/10 vehicle prototype.

Filter (PF) [79], which is a perfect candidate for acceleration on a highly parallel co-processor. The rest of this section is structured in three different parts. The first part describes the architectural choices of our Particle Filter design.

The second part proposes a discussion on how to deploy a fully functional system where the entire application is considered. Besides the PF, the rest of the AV driving software stack is typically deployed on general-purpose cores in heterogeneous FPGA SoCs. This type of partitioning is *per-se* a complex task but also entails repeated movement of data between the host CPU and the accelerator, which hinders performance. *The main objective of this section is to evaluate how the proposed overlay architecture can quickly eliminate undesirable memory transfers among the two subsystems, thanks to a local core to the acceleration logic, which we call the* Proxy Core. The integrated AV design is presented step by step, tackling various practical issues and highlighting how the overlay methodology simplifies the deployment.

An extensive experimental campaign illustrates the various aspects of the performance improvement enabled by hardware acceleration and the Proxy Core compared to the pure SW implementation running on the host cores. Our results show that while FPGA acceleration of PF alone speeds up CPU execution by a factor of 2.5×, employing the overlay and the Proxy Core provides an additional $\approx 2\times$ speedup.

6.1.2 Background

Our target AV software stack is partitioned in a typical *perception-plan-act* design for racing vehicles. The main component is the LiDAR sensor that is used to localize the car in a known map through a method called Particle Filter (PF) [79] (see Figure 6.1). The optimal trajectory of the car is calculated offline via Path Planning [35]. The control loop is closed by a well-known Pure Pursuit [65] algorithm that proves itself an optimal tradeoff between complexity and effectiveness at the speeds our vehicle is expected to race. Figure 6.2 shows the software implementation [79] of our reference Particle Filter in its building blocks. It is composed of five main stages, as follows:

- **Resample**: a high number of particles (i.e., position candidates) is randomly generated;
- **Motion Model**: updates the particles with data from odometry. This data might not be accurate, and typically Gaussian noise is added;
- **Ray Marching**: creates rays[3] centered in each particle, representing their $\{x, y, \theta\}$ poses given the measured ranges from the LiDAR scanner. This step is iterated until rays cover all the map;
- **Compute Weights**: associates a weight to each particle depending on how much it matches the "world" as seen by the LiDAR sensors;
- **Normalize**: flattens the weights within the $[0, 1]$ interval.

The reference implementation was carried out on the application cores of the Xilinx Zynq UltraScale+, which consists of an ARM Cortex-A53 processor (see chapter 2). The figure shows that the performance obtained (4FPS) from the software implementation are not sufficient for use on real-time applications such as autonomous driving (a typical frequency of LiDAR sensors used in the AD domain is at least 40Hz). This first software evaluation shows that the *Ray Marching* phase is the most compute-intensive phase of the algorithm. We leveraged this analysis to design an FPGA-based engine to accelerate the RM computation.

6.1.3 The System Architecture

Figure 6.3 shows our system design employed to accelerate the Particle Filtering algorithm. The proposed overlay design is composed of three main parts: the Ray

[3] The number of rays depends on the LiDAR's resolution (1081 in our Hokuyo UST-10LX).

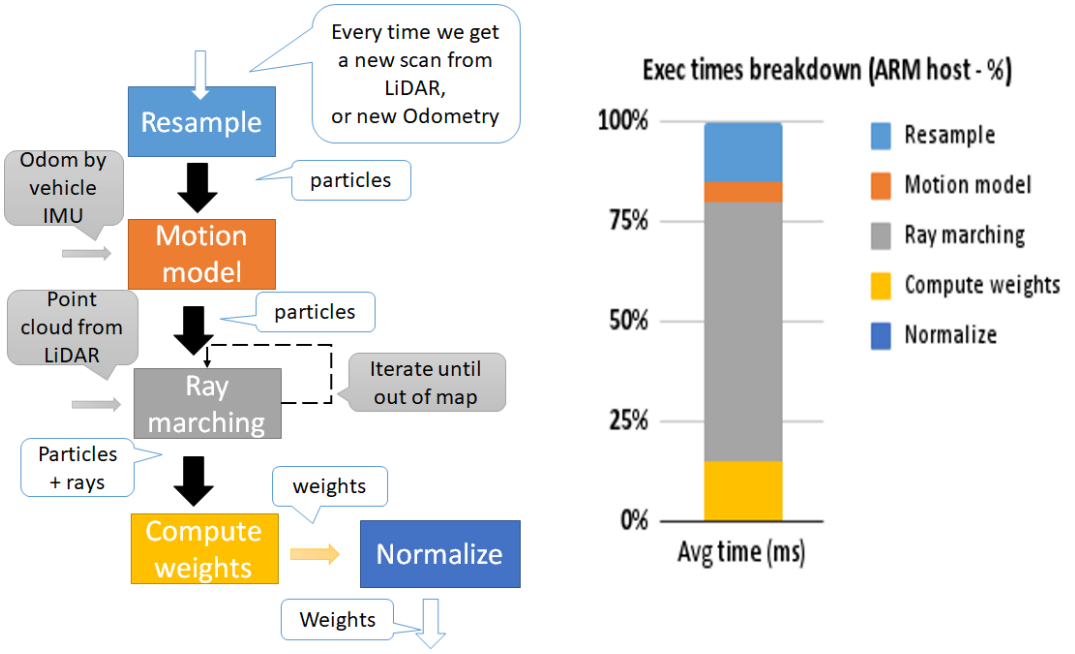


FIGURE 6.2: Scheme and timing breakdown of the reference Particle Filter executed on a Xilinx ZCU102 board.

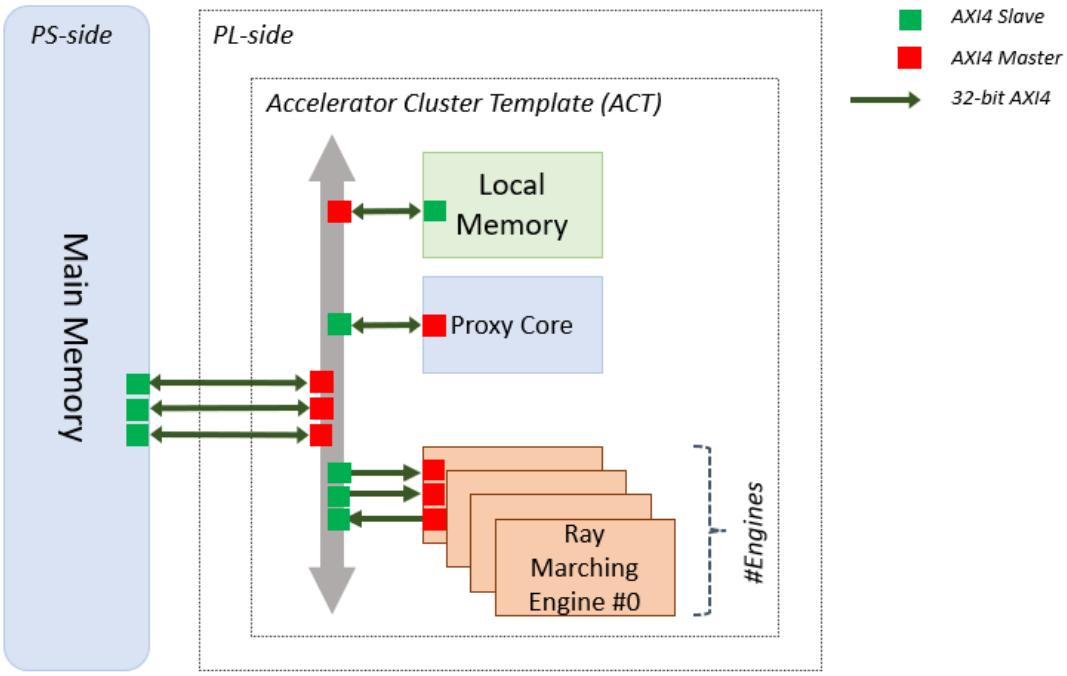


FIGURE 6.3: Proposed FPGA-overlay architecture.

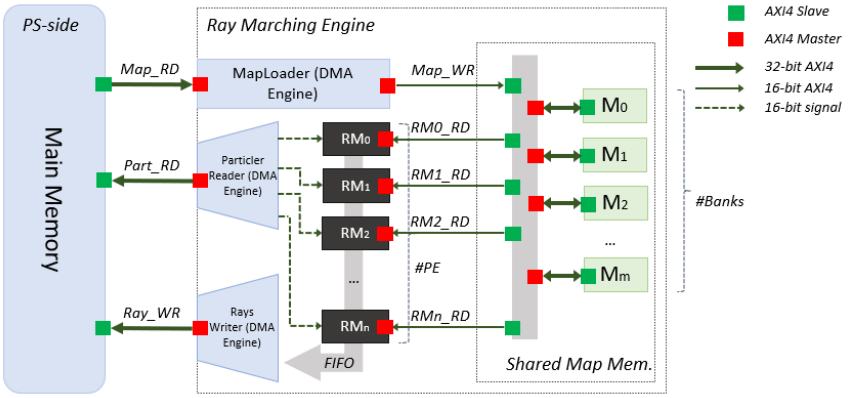


FIGURE 6.4: System organization and Ray Marching Engine internal architecture, acting as co-processor for the PS.

Marching Engine, as described above; a RISC soft-core deployed on the Programmable Logic; and a shared *Scratchpad* memory local in PL, used as data memory for storing required temporary buffers. Figure 6.3 shows how the design is connected to the PS subsystem. Our prototype proxy core is a Xilinx MicroBlaze Microcontroller. The proxy core enables offloading the Compute Weights stage into the Programmable Logic, removing the need to move data out of the Ray Marching. By applying this scheme, the overheads due to memory transfers to/from the DRAM can be avoided. Figure 6.4 shows the internal structure of the Ray Marching Engines (RME), each of which is composed of three different AXI4 interfaces used for data input/output. The main internal components of the engine are *MapLoader* which is the main DMA Engine that pre-loads the map into local memories accessible by the engines. A configurable number of Processing Elements (PE) deals with rays computation for a single particle. *Particle Reader* and *Rays Writer* are two other DMA Engines that respectively take care of loading particles from DRAM and saving the output of the Ray Marching phase.

6.1.4 Evaluation

The proposed system design was evaluated with respect to two distinct implementations: the first one is fully software-based and has been implemented on APU cores (*SW-Only*). In the second version considered (*RME*), only the Ray Marching phase has been accelerated on FPGA, while all the rest of the computation is performed on a Cortex A53. Finally *RME+Overlay* is our solution based on the *overlay* architecture, where we used the RME engine presented above to accelerate the Ray Marching phase, while the *Compute Weights* phase is implemented on top of the

TABLE 6.1: Latency breakdown expressed in milliseconds of Particle Filter Algorithm, varying the number of particles (p).

	SW-Only			RME			RME+Overlay		
p	**60**	**70**	**80**	**60**	**70**	**80**	**60**	**70**	**80**
Resample	0.65	0.99	1.09	0.65	0.99	1.09	0.65	0.99	1.09
Motion Model	0.45	0.52	0.58	0.45	0.52	0.58	0.45	0.52	0.58
Ray Marching	16.82	20.79	22.56	6.15	6.76	6.95	**1.05**	**1.20**	**1.38**
Compute Weights	0.91	1.06	1.24	0.91	0.91	1.24	*1.83*	*2.16*	*2.46*
Normalization	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
Total (ms)	18.83	23.37	25.48	8.16	9.34	9.87	3.98	4.88	5.52
FPS	53.11	42.78	39.25	122.50	107.04	101.33	**250.95**	**204.92**	**181.09**

TABLE 6.2: Resource Usage of different designs

	LUT		LUTRAM		FF	
Board	**Ultra96**	**ZCU102**	**Ultra96**	**ZCU102**	**Ultra96**	**ZCU102**
RME	23473	175540	781	3396	27929	186731
RME+Overlay	/	186430	/	3927	/	197154
Available	70560	274080	28800	144000	141120	548160
%RME	33.27	64.05	2.71	2.36	19.79	34.07
%RME+Overlay	/	68.02	/	2.73	/	35.97

	BRAM		DSP		BUFG	
Board	**Ultra96**	**ZCU102**	**Ultra96**	**ZCU102**	**Ultra96**	**ZCU102**
RME	207	614	58	452	3	3
RME+Overlay	/	667	/	457	/	6
Available	216	912	360	2520	196	404
%RME	**95.83**	67.32	16.11	17.94	1.53	0.74
%RME+Overlay	/	73.14	/	18.13	/	1.49

Proxy-Cores to reduce the memory movements between PS and PL. The results obtained are summarized in Table 6.1. The columns of the table represent different runs characterized by a variable number of particles, respectively 60, 70 and 80. The two accelerated systems require adequate data movements to copy in/out particles and data structures, either to a paged area of the DRAM to properly feed the accelerator. This movement takes approximately 0.03ms. The second design (RME) requires also particles and rays to be copied out from the Ray Marching stage, which takes approximately 30ms. In contrast, our (RME+Overlay) design only needs to move out the results of the full Particle Filter/Normalize stage, i.e., the particle weights. By adopting this design, a further improve has been reached, in terms of end-to-end performance of the Particle Filter by means of$\approx$ 2-3x, compared to a design with the only RME, where data movements occur between the three last steps of the algorithm.

A detailed assessment of the proposed HW/SW partitioning, end-to-end latency (in ms) and throughput (FPS) metrics have been discussed. The latter is representative of the needs of modern AD systems. The LiDAR is the primary sensor for the perception stage, while the former gives an intuition of the reaction time of the

algorithm. Camera-based perception systems are also an interesting option. Still, their effectiveness in the localisation task has not surpassed that of the LiDAR-based ones yet, so they have not been considered in this work. Table 6.2 shows the area occupancy of the proposed design as per the output of the Vivado tools. In the Ultra96 board, where the FPGA resources are limited, the IP alone occupies 96% of the BRAM. There is no room for the Proxy Core. On the larger ZCU102 the complete design with the HW IP and overlay uses 75% of the BRAM. A final important thing to note is that while the Ray Marching IP could be improved, the primary outcome is that the proposed design is completely scalable because the Proxy Core is a completely programmable RISC architecture.

6.1.5 Conclusion

This section presented a scalable design to accelerate the localization of an autonomous robot, with reference to the F1/10th competition as a use case. The main objective of this work was to show the benefits obtainable by adopting an architectural template based on Overlays, where a Proxy-Core, Tightly-Coupled with accelerators similar to what has been presented in the previous chapters of this thesis. **This study corroborates the intuition that on certain applications sacrificing computing performance (the Proxy-Core is slower than the APU) to significantly reduce the necessary memory movements could be convenient**. The complete results achieved can be found in the original paper [7]. Future works in this sense include the integration of RBR design introduced in chapter 5, within the overlay architecture presented in this section to further increase the system predictability.

6.2 CNN-based object Detection and Classification

Building future autonomous CPS, like robots or drones, that are safe and reliable rises a number of challenges, because modern autonomous CPS requires to process a huge amount of data coming from cameras, LiDARs, radars and even the internet, to answer to questions such as: *Where am I? Where is everybody else? How do I get from A to B?*, but also considering autonomous vehicles: *What is the health status of the driver? Do I detect a possible danger for this vehicle, such as a crash with other vehicles or pedestrians?*. The current way of answering these questions employs Deep Neural Networks (DNNs) at their state-of-the-art. DNNs have a tremendous effectiveness

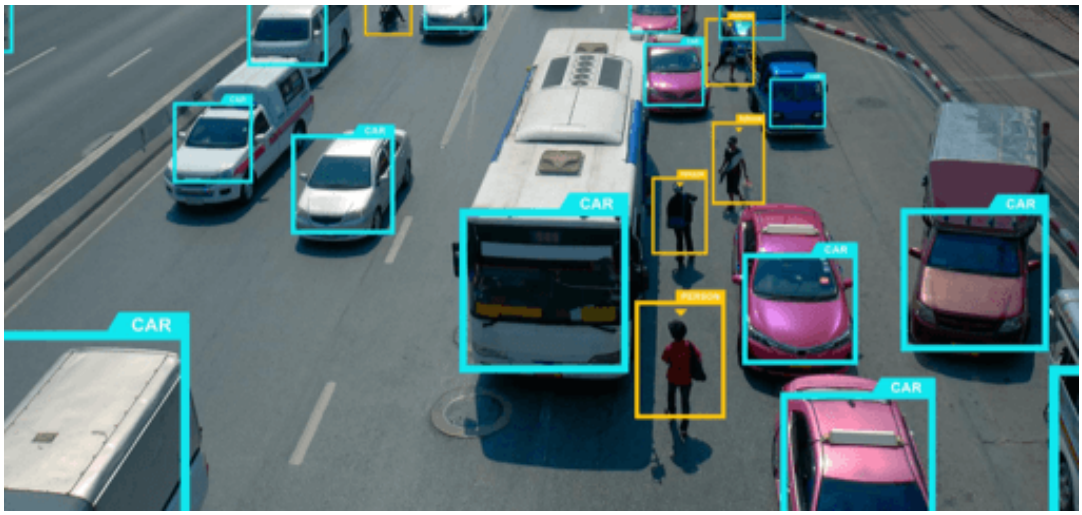


FIGURE 6.5: Example of a Neural Network for objects detection, a typical application for autonomous systems perception.

for tasks as object detection and categorization, which are key modules of the perception and localization systems of tomorrow automated vehicles (Figure 6.5).

The goal of the work presented in this section is to explore and assess the performance, accuracy and power profile of state-of-the-art CNN for image detection and classification, when deployed on embedded computing platforms representative of next-generation CPS domain controllers. Finally, this section is concluded by discussing possible applications of the technologies presented in chapters 4 and 5 in the context of neural accelerators.

6.2.1 Metrics of Interest

The Neural Network deployment on embedded platforms involves the optimization of some aspects based on the considered applications. This represents a tradeoff, because some of the metrics of interest are in contrast each other. The main metrics considered in this work are the following:

1. **Performance**: The main challenge that machine learning engineers are called to solve is to ensure that the latency (i.e. the time required to process a frame) reached by the detector is satisfactory for the considered application. One of the main aspects to take into consideration is the reduced computational power of edge computing systems. In the embedded field, performance optimization involves the use of dedicated accelerators. Often, performance increases causes also an increase on the power consumption of the system;

2. **Energy Efficiency**: Many applications which are based on CNNs (such as autonomous drone navigation) are battery-powered. This requires optimizing the power dissipation of the computing systems in order to both: perform more useful computation while preserving the battery life. This represents a tradeoff with the accuracy of the system as many optimizations involve the use of reduced precision representations or reduced frequency/voltage;

3. **Accuracy**: A key feature of a good detector is the accuracy of its results, to provide reliable predictions. The increase in accuracy is obtained by adopting more complex network models or finer representations, which however often imply a performance degradation.

6.2.2 Neural Network Comparisons

TABLE 6.3: Floating Point versions of YOLO, varying datasets on GPGPU

		Tegra X2			Xavier AGX			Nano		
Model	**Training Set**	**T.put** [FPS]	**mAP** [%]	**Eff** [FPS/Watt]	**T.put** [FPS]	**mAP** [%]	**Eff** [FPS/Watt]	**T.put** [FPS]	**mAP** [%]	**Eff** [FPS/Watt]
YOLOv2	COCO	6.880	23.90	0.775	16.11	23.90	0.597	2.466	23.90	0.365
YOLOv2	VOC	14.30	58.82	2.040	30.80	58.77	1.369	5.691	58.77	0.769
YOLOv2-Tiny	COCO	31.77	5.300	7.334	84.67	5.300	6.805	16.43	5.300	4.373
YOLOv3-Tiny	COCO	75.26	6.480	18.29	201.3	6.480	31.10	23.80	6.540	8.243
YOLOv3	VOC	6.760	78.35	1.642	13.65	78.35	2.109	-	-	-
YOLOv3	COCO	3.430	31.80	0.300	9.380	31.80	0.335	-	-	-
YOLOv3	Berkeley 3	-	-	-	-	-	-	-	-	-
YOLOv3	Berkeley 10	7.690	38.50	0.732	16.25	38.57	0.522	-	-	-

TABLE 6.4: Quantized versions of YOLO, varying datasets on FPGA

		ZU3EG			ZU9EG			VC1902		
Model	**Training Set**	**T.put** [FPS]	**mAP** [%]	**Eff** [FPS/Watt]	**T.put** [FPS]	**mAP** [%]	**Eff** [FPS/Watt]	**T.put** [FPS]	**mAP** [%]	**Eff** [FPS/Watt]
YOLOv2	COCO	-	-	-	-	-	-	-	-	-
YOLOv2	VOC	11.50	63.78	4.416	24.96	65.56	4.571	86.22	65.49	7.660
YOLOv2-Tiny	COCO	-	-	-	-	-	-	-	-	-
YOLOv3-Tiny	COCO	11.97	4.600	9.500	13.89	4.600	6.691	-	-	-
YOLOv3	VOC	6.630	73.17	1.987	14.40	75.07	2.400	42.01	68.16	3.675
YOLOv3	COCO	1.052	31.30	0.696	3.132	31.30	1.366	-	-	-
YOLOv3	Berkeley 3	44.46	19.44	14.82	85.03	19.44	20.30	124.3	15.67	11.46
YOLOv3	Berkeley 10	6.333	55.20	1.898	13.08	55.20	2.516	41.19	19.61	3.620

This section proposes an analysis of different CNNs types to solve the object detection task. The adoption of CNNs to deal with this particular task actually represents the state of the art. The CNNs considered are YOLOv2 and Yolov2-Tiny [57], YOLOv3 and YOLOv3-Tiny [58], YOLOv4 and YOLOv4-Tiny [8] All the CNNs have been trained to perform object detection on different types of datasets, in order to test and assess the same networks in different conditions. The training set used are

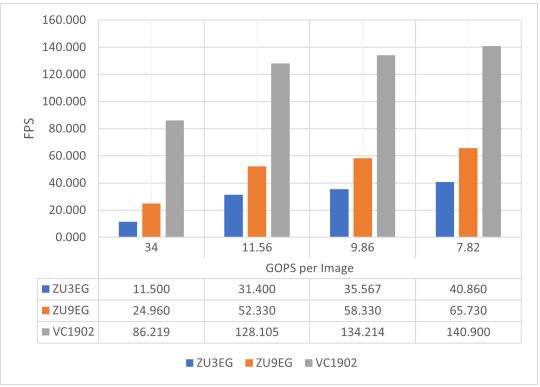
COCO [34], VOC [20] and Berkeley BDD100K[82] that are widely adopted in the literature. After the training phase, the networks has been deployed on the considered platforms: NVIDIA Jetson TX2, NVIDIA Xavier, Xilinx ZU9EG and Xilinx VC1902 (see chapter 2). In addition to the platforms described in the chapter 2, also other platforms are evaluated: the NVIDIA Jetson Nano [48] a platform equipped with 128 CUDA cores of the Maxwell family. The Xilinx ZU3EG is part of the Zynq UltraScale+ family, but compared to the ZU9EG, it features less programmable logic. The ZU3EG can be found at the heart of the Ultra96 evaluation platform. The proposed experimental campaign involves all the metrics of interest mentioned above also considering different configurations of working frequencies.

6.2.3 Lesson Learned

1. **A fair comparison requires effort**: One of the fundamental aspect to consider when comparing different neural networks is to conduct experiments as fair as possible. The primary concern is the network input size, in fact, larger images require more computational resources compared to smaller ones. Another aspect to consider is the model quantization. It is already known that models characterized by more intensive quantizations are more efficient rather than full precision versions. Table 6.3 and 6.4 show the results obtained in terms of throughput, accuracy and power-efficiency, on the considered platforms. To conduct the comparison, different well known training sets have been used. From the point of view of the parameters representation, we leveraged INT8 and FP32 for FPGA and GPGPU respectively, as are typical quantizations for weights and activations that are widely adopted in the scientific literature. In general, as we can see, different networks and datasets affect system performance (throughput), which in turn affects energy efficiency. *There is an in(direct) correlation between dataset and the energy efficiency*. Also, the tables show that FPGA-based (VC1902 in particular) are more power efficient than GPGPUs.

2. **Optimize inference is crucial**: When deploying a neural network on embedded platforms, the inference optimization is an important aspect to take into consideration. Typical optimizations are weights and activations quantizations, a technique that consists of using lower precision parameters (eg. from FP32 to INT8) in order to reduce the latency of the inference phase at the expense of a reduced accuracy [37]. Another promising technique mainly

(A) Throughput



(B) Accuracy

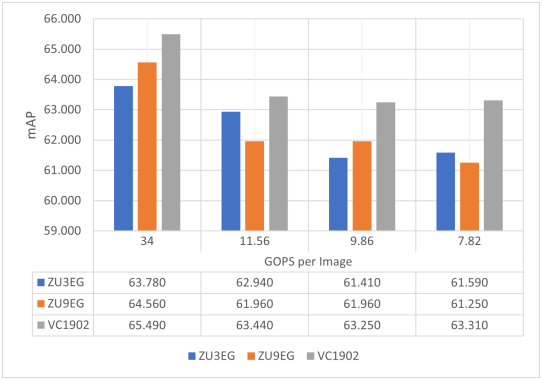


FIGURE 6.6: Comparisons in terms of throughput (FPS) and accuracy (mAP), considering different pruning levels.

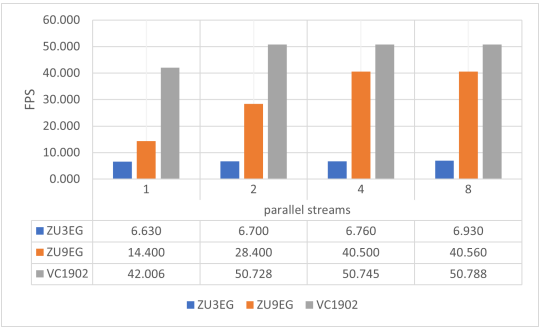
adopted on FPGA-based accelerators is the network pruning [19], that consists of dropping some optionals connections from the neural network model. Also this technique allows to reduce latency at the expense of detection accuracy. The following experiment aims to evaluate the effectiveness of some of the most important optimizations on FPGA-based platforms.

6.2.4 optimizations

Neural network pruning Pruning a neural network means reducing its size and complexity by removing parameters and activations that *"do not play a big role"* inside the whole network [19]. In this case the pruning technique was not evaluated on NVIDIA GPGPUs, because their inference engine framework, does not allow network pruning. Figure 6.6 shows the results. The x-axis shows the reference platforms that are two Zynq UltraScale+ SoCs and one Versal ACAP (see chapter 2). The different bars represents different pruning levels, the smaller is the number the greater is the number of pruned connections.

Multiple parallel flows The most effective/important optimization evaluated in this work consists of loading multiple images into the accelerator, this technique allows to improve the utilization of the accelerator. For example, each layer of the network will have some amount of overhead and synchronization required to compute forward inference. By computing more results in parallel, this overhead is paid off more efficiently. Figure 6.7 highlights the effects of using an increasing number of parallel streams, from one to eight. In general the ZU9EG is the platform which gets the most benefits by adopting parallel streams. This performance increase is due to

(A) YOLOv3



(B) YOLOv4

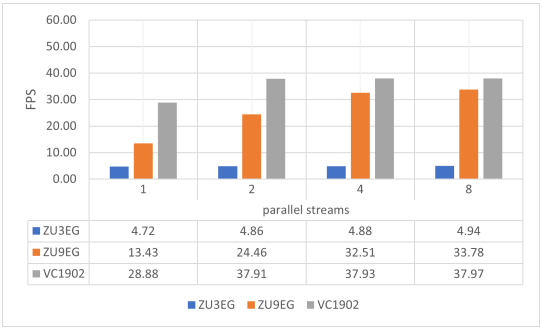


FIGURE 6.7: Benefits in terms of throughput (FPS), when multiple parallel streams are used.

a different hardware organization, since unlike the other two systems, the ZU9EG embeds a greater number of parallel accelerators, which in this case are exploited at their full capacity.

Streaming accelerators In GPGPU systems, the application code must collect a number of data frame (i.e., images from multiple cameras), before being able to pack and offload it to the accelerator. Especially, in NVIDIA systems, the NN package and inference engine drivers give poor freedom to the system designer to access camera in parallel from within the same application. This aspect could certainly can be improved, if the system were not proprietary and closed. A possible enhancement for reconfigurable platforms, and especially Xilinx's SoMs, is that it is possible for engineers to direct the video stream across the FPGA to get through the DRAM banks [29, 76]. This point is really interesting, as it potentially enables on-the-flight detection on the video stream without involving the host cores, and deserves more investigation from system engineers and experts of hardware design.

6.2.5 Conclusion

The aim of this work was to study and evaluate the applicability of CNNs on embedded platforms that are representative of next generation perception engines for autonomous systems. As previously mentioned, neural networks currently represent the state of the art in terms of object detection and classification, two killer applications in the embedded field. As extensively explained in chapter 4, **neural networks are memory-intensive models, which if not carefully regulated could be severely affected by memory interference.** Neural networks also represent perfect candidates to be integrated on the ACT discussed in section 4.3.1. Figure 6.8 shows

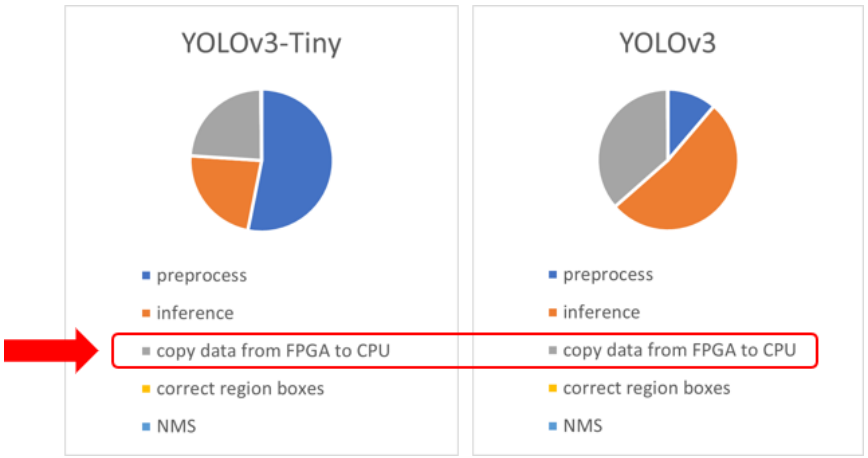


FIGURE 6.8: Breakdown of the different phases a typical neural network is composed. As highlighted memory transfers play a big role on the total latency.

a breakdown considering all the steps needed by a CNN to process an input frame, in general the time needed to copy data back from the FPGA to CPU is comparable with the inference time. **For this reason, performing part of the computation on Proxy-Core in order to reduce memory overheads could bring benefits also in this case.** These aspects will be objects of future investigations. The completed discussion of the achieved results can be found in the original paper [10].

Chapter 7

Conclusion

As introduced in chapter 1 the new generation of heterogeneous systems enabled new applications and use-cases that until a few decades ago were not possible with traditional multicore systems. However, such systems as seen, are typically organized following a *"shared resources"* organization. Architectural template that due to contention on shared resources, could lead to unbounded task execution time increase, that could complicate the adoption of these systems in critical or real-time application domains.

As explained in this thesis, the *memory interference* is less investigated on FPGA-based heterogeneous systems. Furthermore, many memory interference mitigation mechanisms are based on too pessimistic scheduling and time-division policies, which excessively under-exploit the memory bandwidth available on COTS platforms. To overcome these limitations and enhance the state of the art, this thesis proposed the following contributions:

Chapter 4 proposed an analysis of memory interference on two different types of heterogeneous embedded systems, the first based on GPGPU and the second on FPGA, showing that the problem of interference can heavily compromises these two types of systems, resulting in up to 16× of performance degradation. This chapter also presented ideas on how to deploy a HW/SW system inside an FPGA-based Accelerator Cluster Template (ACT) to enable CMRI [16], a recently proposed technique that, using throttling can fine-regulate the memory bandwidth. The proposed mechanism has been used in this chapter as traffic generator to study the memory interference.

Chapter 5 proposed a bandwidth monitoring & regulation mechanism, based on the architectural template already presented in chapter 4. The proposed Tightly-Coupled system fills a gap in the literature, as most of monitoring and control systems are often Loosely-Coupled and platform-specific and fail to provide the required degree of memory bandwidth control. The proposed mechanism (TCMT) has proved to be an effective methodology to mitigate the memory interference problem.

Chapter 6 showed an analysis of two real applications in the embedded domain, both integrated on state of the art heterogeneous systems, characterized by high memory bandwidth requirements. The first concerns an autonomous vehicle localization system, designed using an FPGA-accelerated Particle Filtering algorithm. The system design has been implemented exploiting the ACT presented in chapter 4. The second use case involves the adoption of Convolutional Neural Networks as perception engines for autonomous systems, showing that the ACT and Proxy-Cores can be beneficial also in this case. The emphasis of this chapter is to show, through two use cases, that the technologies proposed in the previous chapters can be effectively applied to implement Cyber Physical Systems that are resilient to memory interference.

7.1 Future research directions

The work presented in this thesis has several points that can be further investigated as future works. The *Runtime Bandwidth Regulator (RBR)* proposed in the chapter 5 allows bandwidth regulation of FPGA tasks only, while the regulation of software tasks running on the host cores is not taken into consideration. The integration of host cores bandwidth regulators, could on the one hand further increase the system bandwidth usage, and on the other hand contribute to the memory interference mitigation. The regulation of software tasks, is carried out in the scientific literature, using software methodologies [16, 83]. Other ways concern the use of Quality-of-Service hardware IPs [67, 86], while other more flexible strategies involve routing the processor cores memory traffic towards the FPGA subsystem [27]. Another aspect that has not been covered in this thesis concerns the I/O devices regulation. The new generation embedded applications are in fact based on the processing of external inputs, which are transmitted to the computing system by means of interfaces to the outside (eg. through the ethernet interface). A possible future work concerns the integration of the proposed regulation mechanisms, also for the I/O devices.

Another important aspect is to integrate the *Runtime Bandwidth Regulator (RBR)* into a scheduling infrastructure that takes into account the system tasks' (FPGA and software) memory bandwidth usage, that is able to guarantee performance to the most critical tasks of the system. The integration of this mechanism on frameworks or libraries of interest to the embedded and real-time community, such as *OpenMP*[1] or *ROS*[2] is also a promising research direction.

[1] https://www.openmp.org/
[2] https://www.ros.org/

Chapter 8

List of Publications

Accepted

1. **G. Brilli**, P. Burgio and M. Bertogna, *Convolutional Neural Networks on Embedded Automotive Platforms: A Qualitative Comparison*, 2018 International Conference on High Performance Computing & Simulation (HPCS), 2018.

2. **G. Brilli** and P. Burgio. 2019. *An open source research framework for IoT-capable smart traffic lights*. In Proceedings of the 5th EAI International Conference on Smart Objects and Technologies for Social Good (GoodTechs '19). Association for Computing Machinery, New York, NY, USA.

3. M. Verucchi, **G. Brilli**, D. Sapienza, M. Verasani, M. Arena, F. Gatti, A. Capotondi, R. Cavicchioli, M. Bertogna and M. Solieri, *A Systematic Assessment of Embedded Neural Networks for Object Detection*, 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2020.

4. **G. Brilli**, A. Capotondi, P. Burgio and A. Marongiu, *Understanding and Mitigating Memory Interference in FPGA-based HeSoCs*, 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2022.

5. A. Bernardi, P. Burgio, **G. Brilli**, A. Capotondi and A. Marongiu, *An FPGA Overlay for Efficient Real-Time Localization in 1/10th Scale Autonomous Vehicle*, 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2022.

Under Review

1. **G. Brilli**, G. Valente, A. Capotondi, T. Di Mascio, P. Burgio, P. Valente and A. Marongiu, *Fine-Grained QoS Control via Tightly-Coupled Bandwidth Monitoring*

and Regulation for FPGA-based Heterogeneous SoCs.

Bibliography

[1] Ahmed Alhammad and Rodolfo Pellizzoni. "Schedulability analysis of global memory-predictable scheduling". In: *2014 International Conference on Embedded Software (EMSOFT)*. 2014, pp. 1–10. DOI: 10.1145/2656045.2656070.

[2] Ahmed Alhammad and Rodolfo Pellizzoni. "Time-predictable execution of multithreaded applications on multicore systems". In: *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2014, pp. 1–6.

[3] Amalthea Consortium. *AMALTHEA. Model Based Open Source Development Environment for Automotive Multi Core Systems*. 2014. URL: https://itea3.org/project/amalthea.html.

[4] ARM. *ARM CoreLink QoS-400 Network Interconnect Advanced Quality of Service*. 2020. URL: https://developer.arm.com/documentation/dsu0026/f/.

[5] Ayoosh Bansal et al. "Evaluating the Memory Subsystem of a Configurable Heterogeneous MPSoC". In: *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*. July 2018, p. 55.

[6] Gianluca Bellocchi et al. "A RISC-V-based FPGA Overlay to Simplify Embedded Accelerator Deployment". In: *2021 24th Euromicro Conference on Digital System Design (DSD)*. 2021, pp. 9–17. DOI: 10.1109/DSD53832.2021.00011.

[7] A. Bernardi et al. "An FPGA Overlay for Efficient Real-Time Localization in 1/10th Scale Autonomous Vehicles". In: *2022 Design, Automation Test in Europe Conference Exhibition (DATE 2022)*. 2022.

[8] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. 2020. arXiv: 2004.10934 [cs.CV].

[9] G. Brilli et al. "Understanding and Mitigating Memory Interference in FPGA-based HeSoCs". In: *2022 Design, Automation Test in Europe Conference Exhibition (DATE 2022)*. 2022.

[10] Gianluca Brilli, Paolo Burgio, and Marko Bertogna. "Convolutional Neural Networks on Embedded Automotive Platforms: A Qualitative Comparison". In: *2018 International Conference on High Performance Computing Simulation (HPCS)*. 2018, pp. 496–499. DOI: 10.1109/HPCS.2018.00084.

[11] Guillem Cabo et al. "SafeSU: an Extended Statistics Unit for Multicore Timing Interference". In: *2021 IEEE European Test Symposium (ETS)*. 2021, pp. 1–4. DOI: 10.1109/ETS50041.2021.9465444.

[12] N. Capodieci et al. "Contending memory in heterogeneous SoCs: Evolution in NVIDIA Tegra embedded platforms". In: *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2020, pp. 1–10. DOI: 10.1109/RTCSA50079.2020.9203722.

[13] Jordi Cardona et al. "Maximum-Contention Control Unit (MCCU): Resource Access Count and Contention Time Enforcement". In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2019, pp. 710–715. DOI: 10.23919/DATE.2019.8715155.

[14] CAST. *Position Paper CAST-32A Multi-core Processors*. Accessed: November 21st, 2021. 2016. URL: https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/media/cast-32A.pdf.

[15] R. Cavicchioli, N. Capodieci, and Mk. Bertogna. "Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms". In: *22nd IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2017, Limassol, Cyprus, September 12-15, 2017*. 2017, pp. 1–10. DOI: 10.1109/ETFA.2017.8247615. URL: https://doi.org/10.1109/ETFA.2017.8247615.

[16] Roberto Cavicchioli et al. "Evaluating Controlled Memory Request Injection to Counter PREM Memory Underutilization". In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2020, pp. 85–105.

[17] F1/10 committee. *The F1/10 autonomous racing competition*. 2015. URL: https://f1tenth.org/.

[18] Cédric Courtaud et al. "Improving Prediction Accuracy of Memory Interferences for Multicore Platforms". In: *2019 IEEE Real-Time Systems Symposium (RTSS)*. 2019, pp. 246–259. DOI: 10.1109/RTSS46320.2019.00031.

[19] Blalock D. et al. "What is the state of neural network pruning". In: *Conference on Machine Learning and Systems* (2020).

[20] M. Everingham et al. "The Pascal Visual Object Classes Challenge: A Retrospective". In: *International Journal of Computer Vision* 111.1 (Jan. 2015), pp. 98–136.

[21] Farzad Farshchi, Qijing Huang, and Heechul Yun. "BRU: Bandwidth Regulation Unit for Real-Time Multicore Processors". In: *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2020, pp. 364–375. DOI: 10.1109/RTAS48715.2020.00011.

[22] Björn Forsberg, Luca Benini, and Andrea Marongiu. "Heprem: A predictable execution model for gpu-based heterogeneous socs". In: *IEEE Transactions on Computers* 70.1 (2020), pp. 17–29.

[23] Björn Forsberg, Luca Benini, and Andrea Marongiu. "Taming Data Caches for Predictable Execution on GPU-based SoCs". In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2019, pp. 650–653. DOI: 10.23919/DATE.2019.8715255.

[24] T. H. Gan et al. "Industry 4.0: Why Machine Learning Matters?" In: *Advances in Condition Monitoring and Structural Health Monitoring*. Ed. by Len Gelman et al. Singapore: Springer Singapore, 2021, pp. 397–404. ISBN: 978-981-15-9199-0.

[25] Giovani Gracioli et al. "A Survey on Cache Management Mechanisms for Real-Time Embedded Systems". In: *ACM Comput. Surv.* 48.2 (Nov. 2015). ISSN: 0360-0300. DOI: 10.1145/2830555. URL: https://doi.org/10.1145/2830555.

[26] Denis Hoornaert, Shahin Roozkhosh, and Renato Mancuso. "A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic". In: *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Ed. by Björn B. Brandenburg. Vol. 196. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 2:1–2:22. ISBN: 978-3-95977-192-4. DOI: 10.4230/LIPIcs.ECRTS.2021.2. URL: https://drops.dagstuhl.de/opus/volltexte/2021/13933.

[27] Denis Hoornaert et al. "Work in Progress: Identifying Unexpected Inter-core Interference Induced by Shared Cache". In: *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2021, pp. 517–520. DOI: 10.1109/RTAS52030.2021.00066.

[28] Intel. *Intel Resource Director Technology*. 2020. URL: https://intel.github.io/cri-resource-manager/stable/docs/policy/rdt.html.

[29] Petar Jokic, Stephane Emery, and Luca Benini. *NN2CAM: Automated Neural Network Mapping for Multi-Precision Edge Processing on FPGA-Based Cameras*. 2021. arXiv: 2106.12840 [eess.IV].

[30] Sagar Karandikar et al. "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)* (2018), pp. 29–42.

[31] Andreas Kurth et al. "HERO: An open-source research platform for HW/SW exploration of heterogeneous manycore systems". In: *Proceedings of the 2nd Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems*. 2018, pp. 1–6.

[32] Edward A. Lee. "Computing Foundations and Practice for Cyber- Physical Systems: A Preliminary Report". In: 2007.

[33] X. Li et al. "A time-multiplexed FPGA overlay with linear interconnect". In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2018, pp. 1075–1080.

[34] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: *Computer Vision – ECCV 2014*. Ed. by David Fleet et al. Cham: Springer International Publishing, 2014, pp. 740–755. ISBN: 978-3-319-10602-1.

[35] Thomas Lipp and Stephen Boyd. "Minimum-time speed optimisation over a fixed path". In: *International Journal of Control* 87.6 (2014), pp. 1297–1311.

[36] Kevin Luttrell, Michael Weaver, and Mitchel Harris. "The effect of autonomous vehicles on trauma and health care". In: *The journal of trauma and acute care surgery* 79 (Sept. 2015), pp. 678–682. DOI: 10.1097/TA.0000000000000816.

[37] Courbariaux M., Bengio Y., and David J.P. "BinaryConnect: Training Deep Neural Networks with binary weights during propagations". In: *Advances in Neural Information Processing Systems (NIPS)* (2015).

[38] K. Manev, A. Vaishnav, and D. Koch. "Unexpected Diversity: Quantitative Memory Analysis for Zynq UltraScale+ Systems". In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. 2019, pp. 179–187. DOI: 10.1109/ICFPT47387.2019.00029.

[39] Emir C. Marangoz, Kyoung-Don Kang, and Seunghee Shin. "Designing GPU Architecture for Memory Bandwidth Reservation". In: *2021 IEEE International*

Symposium on Performance Analysis of Systems and Software (ISPASS). 2021, pp. 87–89. DOI: 10.1109/ISPASS51385.2021.00024.

[40] Jorge Martinez, Ignacio Sañudo, and Marko Bertogna. "Analytical characterization of end-to-end communication delays with logical execution time". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018), pp. 2244–2254.

[41] Peter Marwedel et al. "Survey on Education for Cyber-Physical Systems". In: *IEEE Design & Test* 37.6 (2020), pp. 56–70. DOI: 10.1109/MDAT.2020.3009613.

[42] Alfonso Mascareñas González, Youcef Bouchebaba, and Luca Santinelli. "Multicore shared memory interference analysis through hardware performance counters". In: *10thEuropean Congress on Embedded Real Time Software andSystems(ERTS 2020)*. Toulouse, France, Jan. 2020. URL: https://hal.archives-ouvertes.fr/hal-02446031.

[43] Maxim Mattheeuws et al. "Analyzing Memory Interference of FPGA Accelerators on Multicore Hosts in Heterogeneous Reconfigurable SoCs". In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2021, pp. 1152–1155.

[44] Seung Won Min et al. "Analysis and optimization of I/O cache coherency strategies for SoC-FPGA device". In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2019.

[45] Jan Nowotsch and Michael Paulitsch. "Quality of service capabilities for hard real-time applications on multi-core processors". In: *Proceedings of the 21st International Conference on Real-Time Networks and Systems*. 2013, pp. 151–160.

[46] Jan Nowotsch et al. "Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement". In: *2014 26th Euromicro Conference on Real-Time Systems*. IEEE. 2014, pp. 109–118.

[47] NVIDIA. *Jetson AGX Xavier Developer Kit*. 2018. URL: https://developer.nvidia.com/embedded/buy/jetson-agx-xavier-devkit.

[48] NVIDIA. *Jetson Nano Developer Kit*. 2019. URL: https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

[49] NVIDIA. *Jetson TX2 Module*. 2017. URL: https://developer.nvidia.com/embedded/buy/jetson-tx2.

[50] Matthew O'Kelly et al. *F1/10: An Open-Source Autonomous Cyber-Physical Platform*. 2019. arXiv: 1901.08567 [cs.RO].

[51] Marco Pagani et al. "A Bandwidth Reservation Mechanism for AXI-Based Hardware Accelerators on FPGAs". In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Ed. by Sophie Quinton. Vol. 133. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 24:1–24:24. ISBN: 978-3-95977-110-8. DOI: 10.4230/LIPIcs.ECRTS.2019.24. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10761.

[52] Jeff Parkhurst, John Darringer, and Bill Grundmann. "From Single Core to Multi-Core: Preparing for a new exponential". In: *2006 IEEE/ACM International Conference on Computer Aided Design*. 2006, pp. 67–72. DOI: 10.1109/ICCAD.2006.320067.

[53] Andrea Pellegrini. "Arm Neoverse N2: Arm's 2 nd generation high performance infrastructure CPUs and system IPs". In: *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE. 2021, pp. 1–27.

[54] R. Pellizzoni et al. "Worst case delay analysis for memory interference in multicore systems". In: *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*. 2010, pp. 741–746. DOI: 10.1109/DATE.2010.5456952.

[55] Rodolfo Pellizzoni et al. "A Predictable Execution Model for COTS-Based Embedded Systems". In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. 2011, pp. 269–279. DOI: 10.1109/RTAS.2011.33.

[56] Rahul Rai et al. "Machine learning in manufacturing and industry 4.0 applications". In: *International Journal of Production Research* 59.16 (2021), pp. 4773–4778. DOI: 10.1080/00207543.2021.1956675. eprint: https://doi.org/10.1080/00207543.2021.1956675. URL: https://doi.org/10.1080/00207543.2021.1956675.

[57] Joseph Redmon and Ali Farhadi. *YOLO9000: Better, Faster, Stronger*. 2016. arXiv: 1612.08242 [cs.CV].

[58] Joseph Redmon and Ali Farhadi. *YOLOv3: An Incremental Improvement*. 2018. arXiv: 1804.02767 [cs.CV].

[59] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[60] Francesco Restuccia and Alessandro Biondi. "Time-Predictable Acceleration of Deep Neural Networks on FPGA SoC Platforms". In: *2021 IEEE Real-Time Systems Symposium (RTSS)*. 2021, pp. 441–454. DOI: 10.1109/RTSS52674.2021.00047.

[61] Francesco Restuccia et al. "AXI HyperConnect: A Predictable, Hypervisor-level Interconnect for Hardware Accelerators in FPGA SoC". In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218652.

[62] Francesco Restuccia et al. "Is your bus arbiter really fair? restoring fairness in axi interconnects for fpga socs". In: *ACM Transactions on Embedded Computing Systems (TECS)* 18.5s (2019), pp. 1–22.

[63] Shahin Roozkhosh and Renato Mancuso. "The Potential of Programmable Logic in the Middle: Cache Bleaching". In: *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2020, pp. 296–309. DOI: 10.1109/RTAS48715.2020.00006.

[64] Oriol Sala et al. "SafeTI: a Hardware Traffic Injector for MPSoC Functional and Timing Validation". In: *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. 2021, pp. 1–7. DOI: 10.1109/IOLTS52814.2021.9486689.

[65] Moveh Samuel, Mohamed Hussein, and Maziah Binti. "A Review of some Pure-Pursuit based Path Tracking Techniques for Control of Autonomous Vehicle". In: *International Journal of Computer Applications* 135 (Feb. 2016), pp. 35–38. DOI: 10.5120/ijca2016908314.

[66] Gero Schwäricke et al. "A Real-Time virtio-based Framework for Predictable Inter-VM Communication". In: *2021 IEEE Real-Time Systems Symposium (RTSS)*. 2021, pp. 27–40. DOI: 10.1109/RTSS52674.2021.00015.

[67] Alejandro Serrano-Cases et al. "Leveraging hardware QoS to control contention in the Xilinx Zynq UltraScale+ MPSoC". In: *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2021.

[68] Parul Sohal et al. "E-WarP: A System-wide Framework for Memory Bandwidth Profiling and Management". In: *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2020, pp. 345–357.

[69] Soheil Sohrabi, Haneen Khreis, and Dominique Lord. "Impacts of Autonomous Vehicles on Public Health: A Conceptual Model and Policy Recommendations". In: *Sustainable Cities and Society* 63 (2020), p. 102457. ISSN: 2210-6707. DOI: https://doi.org/10.1016/j.scs.2020.102457. URL: https://www.sciencedirect.com/science/article/pii/S2210670720306776.

[70] Muhammad R. Soliman and Rodolfo Pellizzoni. "PREM-Based Optimal Task Segmentation Under Fixed Priority Scheduling". In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Ed. by Sophie Quinton. Vol. 133. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 4:1–4:23. ISBN: 978-3-95977-110-8. DOI: 10.4230/LIPIcs.ECRTS.2019.4. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10741.

[71] Dharmesh Tarapore et al. "Observing the Invisible: Live Cache Inspection for High-Performance Embedded Systems". In: *IEEE Transactions on Computers* (2021), pp. 1–1. ISSN: 2326-3814. DOI: 10.1109/tc.2021.3060650. URL: http://dx.doi.org/10.1109/TC.2021.3060650.

[72] Michael B. Taylor. "Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse". In: *DAC Design Automation Conference 2012*. 2012, pp. 1131–1136.

[73] U.S. Department of Transportation. *Review of Pending Guidance and Industry Findings on Commercial Off-The-Shelf (COTS) Electronics in Airborne Systems*. Accessed: December 11, 2022. 2001. URL: https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/media/01-41_COTS.pdf.

[74] George Tsamis et al. "Efficient bandwidth regulation at memory controller for mixed criticality applications". In: *2016 11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. 2016, pp. 1–8. DOI: 10.1109/ReCoSoC.2016.7533902.

[75] B. M. Tudor, Y. M. Teo, and S. See. "Understanding Off-Chip Memory Contention of Parallel Programs in Multicore Systems". In: *2011 International Conference on Parallel Processing*. 2011, pp. 602–611. DOI: 10.1109/ICPP.2011.59.

[76] Yaman Umuroglu et al. "FINN". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Feb. 2017). DOI: 10.1145/3020078.3021744. URL: http://dx.doi.org/10.1145/3020078.3021744.

[77] Giacomo Valente et al. "A Composable Monitoring System for Heterogeneous Embedded Platforms". In: *ACM Trans. Embed. Comput. Syst.* 20.5 (July 2021). ISSN: 1539-9087. DOI: 10.1145/3461647. URL: https://doi.org/10.1145/3461647.

[78] *Versal ACAP*. Xilinx. Nov. 2020. URL: https://www.xilinx.com/support/documentation/architecture-manuals/am011-versal-acap-trm.pdf.

[79] Corey H. Walsh and Sertac Karaman. "CDDT: Fast Approximate 2D Ray Casting for Accelerated Localization". In: *Computer Science, Engineering, Mathematics 2018 IEEE International Conference on Robotics and Automation (ICRA)* (2018).

[80] H. Wen and W. Zhang. "Interference Evaluation In CPU-GPU Heterogeneous Computing". In: *IEEE High Performance Extreme Computing Conference (HPEC)* (2017).

[81] Gang Yao et al. "Global real-time memory-centric scheduling for multicore systems". In: *IEEE Transactions on Computers* 65.9 (2015), pp. 2739–2751.

[82] Fisher Yu et al. *BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning*. 2020. arXiv: 1805.04687 [cs.CV].

[83] Heechul Yun 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.

[84] Heechul Yun et al. "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms". In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2014, pp. 155–166. DOI: 10.1109/RTAS.2014.6925999.

[85] Zhe Zhu et al. "Traffic-Sign Detection and Classification in the Wild". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 2110–2118. DOI: 10.1109/CVPR.2016.232.

[86] Matteo Zini et al. "Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms". In: *Software: Practice and Experience* (Nov. 2021). DOI: 10.1002/spe.3053.

[87] *Zynq UltraScale+ Device*. Xilinx. Dec. 2020. URL: https://www.xilinx.com/support/documentation/user%5C_guides/ug1085-zynq-ultrascale-trm.pdf.