TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Computer Engineering, Chair of Adaptive Dynamic Systems

A Modular Platform for Adaptive Heterogeneous Many-Core Architectures

Ahmed Kamaleldin Atef

Born on: 20th May 1990 in Cairo, Egypt

Dissertation

to achieve the academic degree

Doktor-Ingenieur (Dr.-Ing.)

Supervisor and Examiner

Prof. Dr.-Ing. Diana Göhringer (Technische Universität Dresden)

Co-Examiner

Prof. Dr. Ir. Dirk Stroobandt (Ghent University)

Submitted on: 12th April 2023
Defended on: 22nd June 2023

Statement of authorship

I hereby certify that I have authored this document entitled *A Modular Platform for Adaptive Heterogeneous Many-Core Architectures* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. During the preparation of this document I was only supported by the following persons:

Prof. Dr.-Ing. Diana Göhringer

Additional persons were not involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 12th April 2023

Ahmed Kamaleldin Atef

Acknowledgements

First and foremost, I would like to express my deep gratitude to my mentor and supervisor Prof. Dr.-Ing. Diana Göhringer for her guidance and enormous support during my PhD journey. This dissertation would not be possible without her encouragement and advice.

I would also like to thank Prof. Dr. Ir. Dirk Stroobandt, my second examiner, for his feedback and insights which are very helpful to finalize my dissertation. Many thanks to my Fachreferent Prof. Dr. Akash Kumar for his feedback and advice. Many thanks also to the committee members Prof. Dr.-Ing. Horst Schirmeier and Prof. Dr.-Ing. habil. Martin Wollschlaeger.

I would like to deeply thank the entire ADS team at Technische Universität Dresden for the fruitful discussions, collaborations and social activities we had together. I express my deep gratitude to Dr. Lester Kalms, Dr. Ariel Podlubne, Dr. Sergio Pertuz, Gökhan Akgün, Najdet Charaf, Muhammad Ali, Ensieh Aliagha, Veronia Iskandar, and Matthias Nickel, with whom I was collaborating during the period of my PhD.

I am very grateful to my parents, and my brother for always providing me with unlimited support and encouragement.

Abstract

Multi-/many-core heterogeneous architectures are shaping current and upcoming generations of compute-centric platforms which are widely used starting from mobile and wearable devices to high-performance cloud computing servers. Heterogeneous many-core architectures sought to achieve an order of magnitude higher energy efficiency as well as computing performance scaling by replacing homogeneous and power-hungry general-purpose processors with multiple heterogeneous compute units supporting multiple core types and domain-specific accelerators. Drifting from homogeneous architectures to complex heterogeneous systems is heavily adopted by chip designers and the silicon industry for more than a decade. Recent silicon chips are based on a heterogeneous SoC which combines a scalable number of heterogeneous processing units from different types (e.g. CPU, GPU, custom accelerator).

This shifting in computing paradigm is associated with several system-level design challenges related to the integration and communication between a highly scalable number of heterogeneous compute units as well as SoC peripherals and storage units. Moreover, the increasing design complexities make the production of heterogeneous SoC chips a monopoly for only big market players due to the increasing development and design costs. Accordingly, recent initiatives towards agile hardware development open-source tools and microarchitecture aim to democratize silicon chip production for academic and commercial usage. Agile hardware development aims to reduce development costs by providing an ecosystem for open-source hardware microarchitectures and hardware design processes. Therefore, heterogeneous many-core development and customization will be relatively less complex and less time-consuming than conventional design process methods.

In order to provide a modular and agile many-core development approach, this dissertation proposes a development platform for heterogeneous and self-adaptive many-core architectures consisting of a scalable number of heterogeneous tiles that maintain design regularity features while supporting heterogeneity. The proposed platform hides the integration complexities by supporting modular tile architectures for general-purpose processing cores supporting multi-instruction set architectures (multi-ISAs) and custom hardware accelerators. By leveraging field-programmable-gate-arrays (FPGAs), the self-adaptive feature of the many-core platform can be achieved by using dynamic and partial reconfiguration (DPR) techniques.

This dissertation realizes the proposed modular and adaptive heterogeneous many-core platform through three main contributions. The first contribution proposes and realizes a many-core architecture for heterogeneous ISAs. It provides a modular and reusable tile-based architecture for several heterogeneous ISAs based on open-source RISC-V ISA. The modular tile-based architecture features a configurable number of processing cores with different RISC-V ISAs and different memory hierarchies.

To increase the level of heterogeneity to support the integration of custom hardware accelerators, a novel hybrid memory/accelerator tile architecture is developed and realized as the second contribution. The hybrid tile is a modular and reusable tile that can be configured

at run-time to operate as a scratchpad shared memory between compute tiles or as an accelerator tile hosting a local hardware accelerator logic. The hybrid tile is designed and implemented to be seamlessly integrated into the proposed tile-based platform.

The third contribution deals with the self-adaptation features by providing a reconfiguration management approach to internally control the DPR process through processing cores (RISC-V based). The internal reconfiguration process relies on a novel DPR controller targeting FPGA design flow for RISC-V-based SoC to change the types and functionalities of compute tiles at run-time.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ACK Acknowledgement

ALU Arithmetic Logic Unit

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

ASIP Application Specific Instruction Set Architecture

AXI Advanced Extensible Interface

BRAM Block Random-Access Memory

BW Bandwidth

CAD Computer-Aided Design

CLB Configurable Logic Block

CLK Clock

CMOS Complementary Metal–Oxide–Semiconductor

CNN Convolutional Neural Network

CPU Central Processing Unit

CU Compute Unit

DDR Double Data Rate

DFG Data Flow Graph

DMA Direct Memory Access

DPR Dynamic Partial Reconfiguration

DPU Data Processing Unit

DRAM Dynamic Random-Access Memory

DSA Domain Specific Accelerator

DSE Design Space Exploration

DSP Digital Signal Processor

DTCM Data Tightly Coupled Memory

DVFS Dynamic Voltage and Frequency Scaling

eFPGA Embedded FPGA

FF Flip-Flop

FFT Fast Fourier Transform

FIFO First-in-First-out

FPGA Field-Programmable Gate Array

FSM Finite State Machine

GALS Globally Asynchronus Locally Synchronus

GCC GNU C Compiler

GPU Graphics Processing Unit

HBM High Bandwidth Memory

HDL Hardware Description Language

HLS High-Level Synthesis

HMC Hybrid Memory Cube

HPC High-Performance Computing

I/O Input/Output

ICAP Internal Configuration Access Port

IFM Input Feature Map

ILP Instruction-Level Parallelism

IoT Internet of Things

IP Intellectual Property

ISA Instruction Set Architecture

ITCM Instruction Tightly Coupled Memory

LCA Loosely Coupled Accelerator

LLC Last Level Cache

LUT Lookup Table

MAC Multiply and Accumulate

MPI Message Passing Interface

MPSoC Multiprocessor System-on-Chip

NI Network Interface

NoC Network-on-Chip

NPU Neural Processing Unit

NUMA Non-Uniform Memory Access

OFM Output Feature Map

OpenCL Open Computing Language

OpenMP Open Multi-Processing

OPS Operations per Second

OS Operating System

PCAP Processor Configuration Access Port

PCI-E Peripheral Component Interconnect Express

PE Processing Element

PLM Private Local Memory

PRR Partial Reconfigurable Region

PU Processing Unit

QNN Quantized neural Network

QoS Quality of Service

RISC Reduced Instruction Set Computer

RM Reconfigurable Module

RP Reconfigurable Partition

RTL Register Transfer Level

SAR Synthetic Apereture Radar

SDK Software Development Kit

SIMD Single Instruction Multiple Data

SoC System-on-Chip

SPI Serial Peripheral Interface

SRAM Static Random-Access Memory

TCA Tightly Coupled Accelerator

TCM Tightly Coupled Memory

TLB Transaction Lookaside Buffer

UART Universal Asynchronous Receiver-Transmitter

UAV Unmanned Aerial Vehicle

UMA Uniform Memory Access

URAM Ultra Random-Access Memory

VHDL Very High Speed Integrated Circuit Hardware Description Language

VLSI Very Large-Scale Integration

WCET Worst-Case Execution Time

1 Introduction

1.1 Motivation

Over the last decade, CMOS technology scaling has slowed down leading to the end of the historical correlation between performance and energy efficiency improvements and CMOS scaling process [1]. Currently, energy dissipation is the main limiting factor for the performance of processing units (e.g. CPUs, microprocessors) which is due to the constant power density resulting from CMOS technology scaling. Consequently, computing performance scalability cannot be further improved by frequency scaling due to the crisis of the power wall. In parallel, the ever-growing size of data processing like in big data and machine learning domains increases the challenges to achieve scalable computing performance with nominal energy efficiency by just increasing the transistor counts on the chip [2]. Therefore, a shift in computing paradigms has been started by replacing single-core designs with multi-core and many-core architectures by exploiting computing parallelism techniques to achieve a substantial computing speedup with an improvement in energy efficiency.

The paradigm shift in computing-centric architectures from single-core to multi-/many-core systems leads to the emergence of heterogeneous system-on-chip (SoC) designs. Where multiple heterogeneous components are integrated on the same silicon chip. The evolution of compute-centric architectures, trying to overcome the slowdown of Moore's law, opens the door for domain-specific architectures by tailoring and customizing the computing units to efficiently execute a specific scientific algorithm or an application domain using custom hardware accelerator units. On the other hand, heterogeneous multi-instruction set architectures (multi-ISA) such as the recent generation of heterogeneous CPUs with little and big cores architectures are able to find the best tradeoff between computing performance and energy efficiency for general-purpose workloads based on runtime workload's kernels computing requirements [3], [4]. Moreover, techniques such as digital-voltage-frequency-scaling (DVFS) and near-threshold computing are currently adopted for recently developed heterogeneous mobile SoC to have scalable computing performance with a low power budget.

In parallel, recent initiatives towards agile hardware development open-source tools and microarchitecture aim to democratize silicon chip production for academic and commercial usage [5]. Agile hardware development aims to reduce development costs by providing an ecosystem for open-source hardware microarchitectures and hardware design processes. One of the main contributions towards agile hardware development is RISC-V open-source ISA, where a series of open-source RISC-V-based microprocessors are royalty-free [6] that can be used directly in the development of SoC designs as off-shelf IP cores or through customization by adding application-specific custom instructions for a new RISC-V ISA extension.

In addition, open-source designs and development tools are aiming to create open-source SoC design frameworks, where SoC development and customization will be relatively less complex and less time-consuming than conventional design process methods [7], [8]. In comparison with baseline SoC design and development, agile hardware development provides the ability to seamlessly integrate a new hardware module (e.g. custom hardware accelerator, microprocessor with special ISA extension) within a unified and modular SoC architecture. Hence, domain-specific accelerators with agile hardware development contributions open the market for more new players to develop their own hardware accelerator intellectual properties (IPs) without the necessity to have strong expertise in SoC design and integration [9].

Multi- and many-core architectures are the dominant type of architecture for the current SoC designs [10]. They feature a scalable and heterogeneous number of processing units, where interconnection and communication between architectural units are crucial to maintain the desired computing performance. Many particular challenges are arising regarding communication between different processing units, system-level scalability, programmability, and last but not least the need for a unified platform or an integration methodology between heterogeneous compute units. More specialization by heterogeneity decreases the system-level scalability and flexibility due to the inherent structure of heterogeneous processing units and custom hardware accelerators. Therefore, several recent approaches focus on tile-based or clustered architectures that can offer more degree of scalability than traditional multi-core processors. The design and development of scalable many-core heterogeneous architectures is a cumbersome process due to several system-level challenges regarding integration and interaction between a large number of non-unified heterogeneous processing units. As a result, the design process is always accompanied by continuous inflation in design costs and a limited degree of post-design upgrades.

This dissertation enables the design of scalable many-core SoC designs with a regular and flexible architecture that hides the complexities of heterogeneous many-core integration for rapid prototyping and low-cost generation of multiple heterogeneous many-core taxonomies using modular and reusable tile-based computing units.

In order to provide a modular and agile many-core development methodology, this dissertation proposes a development platform for heterogeneous and adaptive many-core architectures consisting of a scalable number of heterogeneous tiles that maintain design regularity features while supporting heterogeneity. The proposed platform hides the integration complexities by supporting modular tile architectures for general-purpose processing cores and custom hardware accelerators. In addition, the communication between heterogeneous compute tiles is conducted through a unified communication model through a generic NoC architecture [11]. The proposed many-core platform promotes architectural components reuse and guarantees hardware portability across different many-core taxonomies designs. The platform exploits the regularity of compute tiles and processing element architectures to support the seamless integration of new compute units based on target application requirements or future design upgrades.

By leveraging field-programmable-gate-arrays (FPGAs), the self-adaptive feature of the many-core platform can be achieved by supporting dynamic and partial reconfiguration techniques. In this dissertation, a novel reconfiguration management unit is proposed to internally control the DPR process from a permanent compute tile to configure the many-core architecture at runtime in terms of type and number of heterogeneous compute tiles. The self-adaptive

feature allows the deployment of different many-core taxonomies based on changing application requirements at runtime. Also, It allows further design upgrades without the need to repeat the design and generation process for the upgraded many-core design. Therefore, design modularity and adaptability are keys for reducing design and integration costs and promoting the commodity of many-core architectures for emerging application domains.

1.2 Objective of this Dissertation

The presented background of the research field in the previous section brings us to discuss the current challenges and the main focus of this dissertation. As presented at the beginning, the degree and level of heterogeneity come on top of the outstanding challenges in many-core SoC architectures, more noticeably, in the existence of the domain-specific accelerator era which is increasing the level of heterogeneity and scaling complexities of many-core SoC architectures. However, the effects of the emergence of new open-source ISAs (e.g., RISC-V ISAs) and how to leverage its existence with custom hardware accelerators in many-core SoC architectures is yet an open research point [12].

In heterogeneous many-core systems, integration and communication between a scalable number of heterogeneous processing cores lead to several system-level challenges that increase the design effort and costs as well as limited micro-architecture post-design upgrades. Hence, new approaches towards the design of agile many-core SoC architectures have recently flourished [13]. The motivation is to provide a modular and agile many-core system with flexibility and reusability features to support different micro-architecture configurations as well as post-design incrementation with new heterogeneous components (e.g. new ISAs, custom hardware accelerators). Therefore, the design and development of agile many-core systems start from the tile micro-architecture. The many-core system is based on a tile-based architecture connected through a NoC to keep the needed scalability and high communication bandwidth between the tiles. The main concern is to realize reusable and flexible tile architecture types that can be configured or augmented with new heterogeneous components at run-time.

Accordingly, modular micro-architectures to support multiple-memory hierarchies and seamless integration of several heterogeneous components are still open wide for research. It is further worth mentioning that, the recent slowdown of CMOS scaling technology and the end of Dennard scaling will lead to more architectural specialization and an extreme level of heterogeneity that requires the design and deployment of largely fixed-function accelerators based on an algorithm or application requirements [14]. Thus highlighting the need for an agile many-core architecture design to cope with new computing challenges.

On the other hand, several architectural solutions are presented in the literature targeting reconfigurable and adaptive SoC. However, existing literature solutions are not dedicated to RISC-V-based SoC. Therefore, in dealing with the self-adaptive feature for many-core systems, an internal reconfiguration management unit for RISC-V-based SoC is developed. It can change the internal functionality or the configuration of tiles micro-architecture as well as custom hardware accelerator logic.

In that context, the work in this thesis deals with these untackled research points combined, micro-architecture modularity, higher level of heterogeneity, and adaptability, by focusing on how to realize a modular and adaptive many-core SoC architecture for multi ISAs and

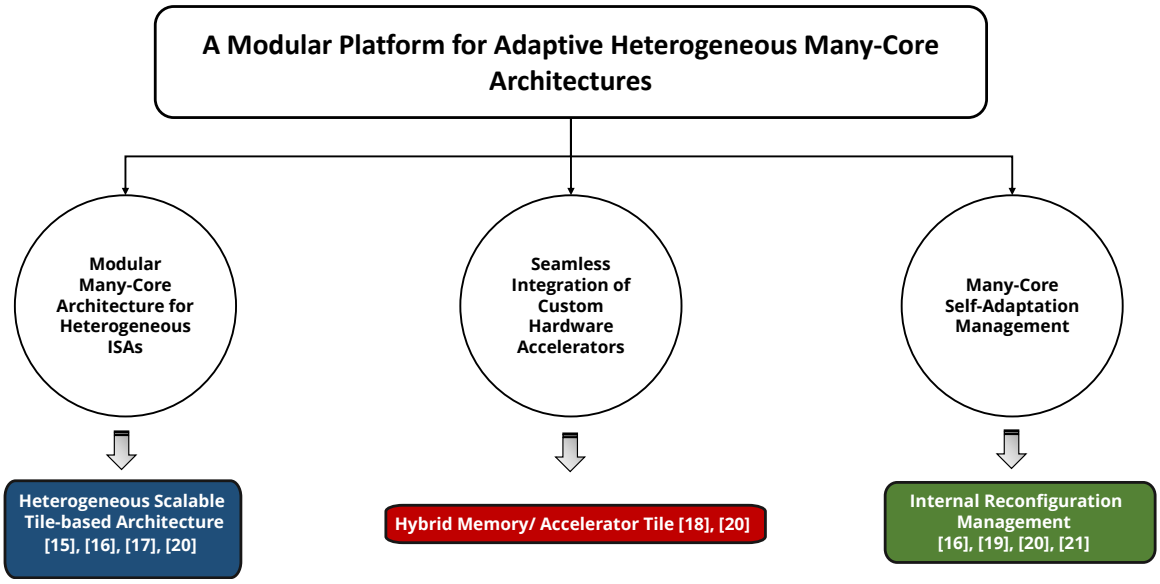


Figure 1.1: Main contributions towards the realization of a modular platform for adaptive heterogeneous many-core architectures.

seamless integration of heterogeneous custom hardware accelerators at run-time. The dissertation provides a thorough investigation of heterogeneous many-core architectures by implementing a modular and adaptive tile-based many-core architecture for heterogeneous ISAs and custom hardware accelerators.

The thesis presents a modular and configurable tile-based architecture with several types of tile architecture where tiles can host (1) a configurable multi-core architecture based on several heterogeneous ISAs [15], [16] with different memory hierarchies [17], (2) custom hardware accelerators and shared memory blocks through a hybrid tile architecture to leverage the reusability of architectural components [18], [19], [20]. Several signal processing use cases accelerators have been used for evaluation. Further, internal run-time reconfiguration management is developed and implemented to leverage self-adaptability for the proposed tile-based many-core architecture [21], [16]. The main contributions of this dissertation are presented in the following section.

1.3 Own Contributions

The contributions of this doctoral thesis are as follows: (1) Modular many-core architecture for heterogeneous ISAs, (2) Seamless integration of custom hardware accelerators through a hybrid memory/accelerator tile architecture, and (3) Many-core runtime reconfiguration management through an internal reconfiguration management system. This is illustrated in Figure 1.1 which presents the three main contributions and how they contribute towards the realization of a modular platform for adaptive heterogeneous many-core architectures.

The first contribution is based on state-of-the-art analysis for processor-centric many-core architectures that directs to the necessity for a modular and reusable many-core platform to support heterogeneous ISAs with different architectural configurations for ever-increasing computing demands. As a result, a modular tile-based many-core architecture for several

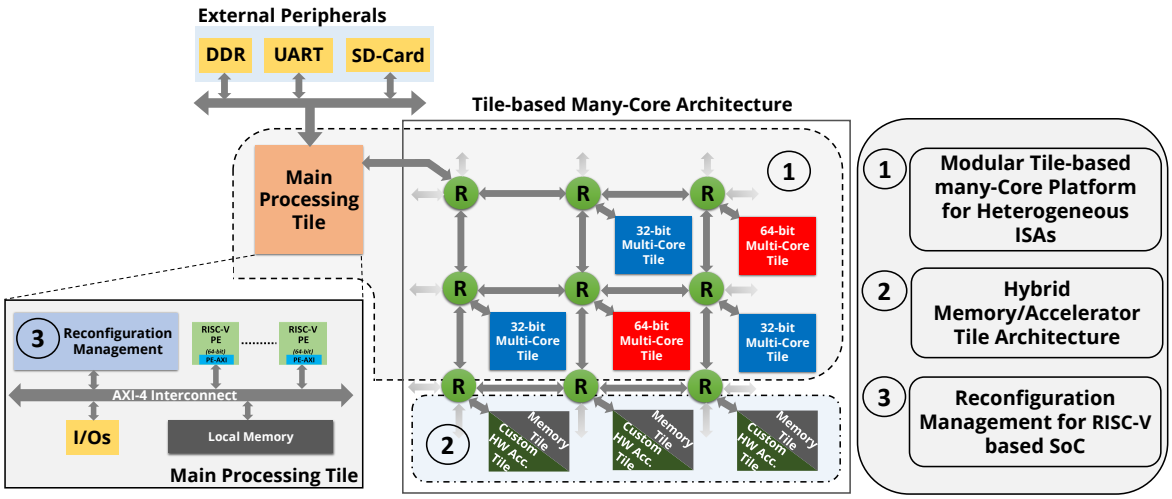


Figure 1.2: Proposed adaptive and modular many-core architecture including main dissertation contributions: (1) modular tile-based for heterogeneous ISAs, (2) hybrid architecture tile for custom hardware accelerators and memory blocks, and (3) reconfiguration management unit for self-reconfigurable RISC-V-based SoC.

heterogeneous ISAs is proposed. In order to increase the level of heterogeneity and support seamless integration of custom hardware accelerators and memory modules for domain-specific applications, a hybrid memory/accelerator tile architecture is proposed.

Therefore, by combining the first and second contributions, the proposed many-core platform by this doctoral thesis supports an unprecedented level of heterogeneity with flexible architectural configurations using modular and reusable tile architectures. Further, the proposed modular many-core architecture is occupied with an internal reconfiguration management unit as the third contribution for self-adaptive purposes. The internal reconfiguration management unit is responsible for changing tiles functionalities and configurations during run-time through DPR. The proposed contributions are shown in Figure 1.2 shaping the proposed adaptive and modular many-core architecture proposed by this doctoral thesis. The main highlights of each contribution are presented in the following.

- Modular Tile-based Many-Core Architecture for Heterogeneous ISAs [15], [16], [17], [20].
  On the level of heterogeneous ISA designs, this dissertation worked on the gap of a missing modular many-core platform to support multiple heterogeneous ISAs. The proposed modular many-core platform features a scalable tile-based architecture where each tile can host a single or a multi-core architecture with different RISC-V ISA-based PEs [15], [16]. Each tile supports different memory configurations for shared and local instruction/data scratchpad memories associated with multi-core or single-core configurations [17]. Furthermore, the proposed many-core platform supports multiple communication models for data sharing and transmission between heterogeneous tiles through a scalable NoC architecture. A unified programming method is developed to target multiple RISC-V ISAs for 64- and 32-bit architectures. The proposed many-core platform supports FPGA design flow for hardware evaluation in terms of resource utilization and power consumption. Further, computing scalability and performance are evaluated in terms of achievable operations per second and memory bandwidth for several many-core configurations using multiple signal processing-based use cases.

- Hybrid Memory/Accelerator Tile Architecture for Tile-based Many-Core Systems [18], [20].
  Hybrid memory/accelerator tile architecture is proposed as the outcome of studying accelerator-centric architecture designs. It supports two modes of tile operation as a memory tile or an accelerator tile hosting a custom hardware accelerator using a modular tile architecture [18], [20]. The tile supports the seamless integration of custom hardware accelerators to the proposed many-core platform through the modular hybrid tile architecture. Furthermore, leveraging the hybrid tile architecture to support noncoherent memory sharing between custom hardware accelerators and heterogeneous ISAs tiles. Multiple hardware accelerators from the signal processing domain are developed and used as use cases for evaluation.

- Reconfiguration Management for Self-Adaptive Tile-based Systems [16], [19], [20], [21].
  A reconfiguration management unit is proposed to allow self-adaptation for the proposed tile-based many-core system [19]. The applied self-adaptation approach is based on self-controlling and management of the reconfiguration process through a main processing tile [21]. The internal reconfiguration process relies on a novel DPR controller targeting FPGA design flow for RISC-V-based SoC to change the types and functionalities of many-core tiles at run-time [16], [20]. Furthermore, the performance of the proposed reconfiguration management unit is evaluated based on hardware resource utilization, maximum achievable reconfiguration throughput and power consumption. The proposed reconfiguration management achieves a faster reconfiguration time compared to state-of-the-art DPR-based reconfigurable SoC.

1.4 Structure of this Dissertation

This dissertation is structured into six chapters including this one, organized as follows.

Chapter 2 presents the preliminary background and literature review of the state-of-the-art in the field of this dissertation covering topics of heterogeneous many-core architectures and adaptive SoC architectures. The chapter discusses the current research directions for heterogeneous many-core architectures including processor-centric and accelerator-centric approaches. In addition, several tile-based many-core architectures are reviewed in order to explore state-of-the-art many-core realization techniques for heterogeneous ISAs and custom hardware accelerators. The chapter is concluded by discussing open research directions that this dissertation aims by bridging adaptive computing, computing heterogeneity with many-core architectures. Further, the contributions of this dissertation are presented and positioned within the presented state-of-the-art.

Chapter 3 presents the first proposed contribution of a modular tile-based many-core architecture for heterogeneous ISAs.The chapter starts by presenting a modular tile architecture that can host multiple numbers and types of PEs based on different RISC-V ISAs with shared and local scratchpad memories. Multiple RISC-V-based PEs are presented followed by different supported interfaces to be integrated within the tile architecture. System scalability and communication models between tiles are then presented using a parametrized NoC architecture. The programming method and software execution are later presented supporting 32-/64-bit programming flows. The chapter is finally concluded with a brief summary and

discussion of the proposed tile-based many-core architecture. The content of this chapter is based on the following published work: [15], [16], [17], [20].

Chapter 4 presents the second proposed approach to support a hybrid memory/accelerator tile within the proposed tile-based many-core architecture in the previous chapter. The chapter starts by presenting the architectural components of the proposed approach and the seamless integration method of RTL/HLS-based hardware accelerator to the tile. Then, tile external interfaces, and integration to the other compute tiles are presented, followed by a description of the control and data messages over the NoC for communication with other compute tiles. Hardware and experimental results are then presented for the proposed hybrid tile using several use cases from the signal processing domain. The content of this chapter is based on the following published work: [18], [20]

Chapter 5 presents the third proposed approach to support run-time reconfiguration through an internal reconfiguration management unit. The proposed approach relies on the development of an internal reconfiguration manager suitable for RISC-V-based SoC to be inserted within the main processing tile of the proposed many-core architecture. The reconfiguration process is based on dynamic partial reconfiguration for FPGAs. The chapter starts by presenting the internal hardware architecture of the proposed reconfiguration management unit. It then presents software management and abstraction layer to control and manage the reconfiguration from RISC-V-based PEs in the main processing tiles including partial bitstream transfer from external memory storage to FPGA configuration memory. The performance and hardware results of the proposed reconfiguration management unit are then presented and discussed with several many-core configuration scenarios. The content of this chapter is based on the following published work: [16], [19], [20], [21].

Chapter 6 summarizes and concludes this dissertation and presents future work insights.

2 Background and State-of-the-Art

This chapter provides the essential background information for the following chapters based on the current state-of-the-art. For the design and exploration of modular and adaptive many-core architectures, this dissertation covers two main research topics. The first part discusses state-of-the-art tile-based many-core architectures based on system architecture, degree of heterogeneity, and hardware accelerators integration as presented in Section 2.1 and Section 2.2. The second part explores adaptive computing systems in order to achieve a self-adaptive many-core system. Therefore, a comprehensive overview of adaptive computing platforms and reconfiguration management frameworks are presented in Section 2.3. Section 2.4 presents the contribution of this dissertation and the comparison to the state-of-the-art towards the realization of a modular and adaptive many-core system. Finally, the chapter is summarized in Section 2.5.

2.1 Tile-Based Many-Core Architectures

The end of Dennard scaling started to appear around the year 2005 [22]. Single-core processor chips start to hit the power density limit and therefore single-threaded performance began to slow down. Therefore, the semiconductor industry had started to find a new computing paradigm that could keep the continuity of Moore's law and the growth of technology scaling. Therefore, workload parallelism could improve computing performance through multi-core processing architectures, driven by lower frequencies with less power-hungry pipelines. Each core can support single or more threads of execution so that the total number of instructions per cycle can increase with the growing number of available cores per chip, which tends to reduce the overall performance per watt and keeps the power density under a certain limit based on technology nodes [23].

Degrees of workload parallelism are affected by several system-level factors, such as types of supported memory hierarchy, inter-core interconnect topologies, and parallel programming methods. System-level factors contribute towards setting an upper bound to the nominal performance obtained from multi-core architectures. Such system-level factors are correlated with Amdahl's law to determine upper-bound variations on multi-core architectures [24]. Nevertheless, multi-core architectures are being at the centre of the compute-centric paradigm for a decade. Compute-centric architectures constitute the majority of current computing machines from embedded domains up to high-performance computing systems. Compute-centric architectures have witnessed a tremendous evolution in the field of computer architecture and embedded systems [25], evolving from single-core architectures to hundreds of cores SoC [26].

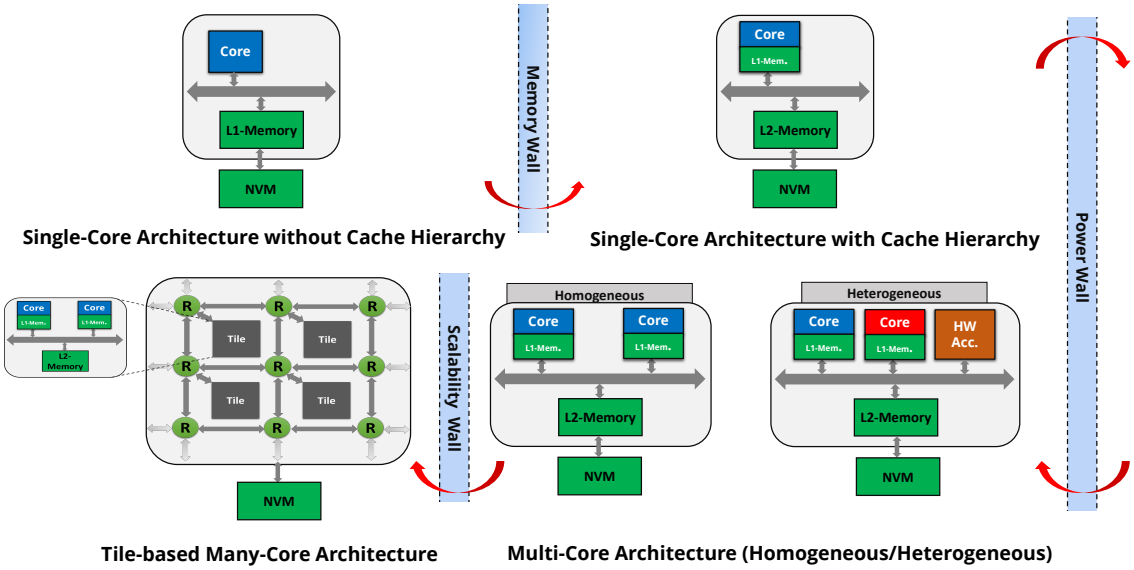


Figure 2.1: Evolution of compute-centric systems from single-core architectures towards tile-based many-core architectures [10].

Currently, a variety of homogeneous and heterogeneous multi-core and many-core architectures are leveraged in mainstream chips. Many-core architectures consist of a large number of cores with more sophisticated memory hierarchies and interconnect compared to typical multi-core systems. Compute performance scaling and low power consumption have been ubiquitous and continual problems for computer architectures throughout its history. Moreover, several intertwined challenges related to efficient programming, limited memory bandwidth and data locality inherited from Von Neumann architecture represent main motivations for compute-centric architecture improvements [27].

Figure 2.1 shows the evolution steps of compute-centric architectures from single-core to tile-based many-core architectures, including major walls of computer architecture [10]. Processor performance kept increasing rapidly, while memory latency to processor computing latency is significantly slower. Therefore, the memory wall is the first computer architecture challenge that had to be overcome by leveraging and optimizing cache hierarchies and bringing data as close as possible to processors [28]. However, cache-unfriendly data structures cannot be handled well by caching as the disparity rapidly increases [29]. On the other hand, power dissipation and energy consumption kept increasing with further improvement of clock frequencies for single-core architectures to achieve higher compute performance along with increasing design complexity.

The power wall describes this obstacle of computing performance scaling as mentioned in Figure 2.1. Therefore, increasing compute performance scaling requires a shift in compute-centric architecture design to overcome the power wall. As a result, different concepts of multi-core processors have been introduced over the last two decades since the end of Dennard scaling, aiming to achieve scalable compute performance with higher performance per watt. Multi-core processor architectures are typically split into two major architectural groups based on the type of processing cores. First, homogeneous architectures consist of the same type of processing cores connected through a communication fabric (e.g. shared bus, NoC) with memory-mapped I/O peripherals. The second type is heterogeneous architectures

with several types of processing cores and custom hardware accelerators. Heterogeneous multi-core architectures are a big leap in the history of the evolution of compute-centric architectures [30]. Currently, multi-ISA heterogeneous multi-core are increasingly adopted [31] combining large high-performance cores and small power-efficient ones for general-purpose mixed workloads. Additionally, incorporating custom hardware accelerators is increasingly used to improve overall efficiency by employing specialization in current multi-core architectures supporting domain-specific workloads.

Parallel programming for multi-core architectures is closely tied to the system's memory organization, which can be classified into centralized shared memory, distributed shared memory, and fully distributed shared memory. In shared memory architectures, all processing cores share the same address space of the memory subsystem and the memory bandwidth is shared between cores. Memory bandwidth is an important factor for the performance of multi-core architectures but limits the degree of scalability. The shared memory programming model relies on fine-grained data sharing and dynamic memory access behavior that can be handled by compilers. However, to avoid race conditions programmers need to manage synchronization efficiently between cores [32].

Several parallel programming standards are provided to manage data movements between cores and memory. For example, OpenMP [33] is an industry-standard that can be used to ease parallel programming of shared memory architectures. In contrast, message-passing models are used for distributed memory architectures that do not provide a shared address space. Therefore, communication between cores needs to be established by a message-based communication model. Message passing interface is a well-known standard library for distributed memory multi-core architectures, which includes a full range of message-passing primitives [34]. Despite the tackled computing challenges by conventional homogeneous and heterogeneous multi-core architectures, system scalability becomes more complex and becomes a burden to increase the level of heterogeneity for data-centric workloads. Besides, more specialization by heterogeneity decreases the system-level flexibility due to the inherent structure of heterogeneous processing cores and accelerators [35]. Therefore, several recent approaches focus on tile-based or clustered architectures that can offer more degree of scalability than traditional multi-core processors [36].

As shown in Figure 2.1 tile-based architectures overcome the scalability wall by providing a distributed scaled number of heterogeneous compute tiles that can host shared or distributed multi-core architectures inside. However, traditional multi-core architectures can be programmed easily, as current commodity parallel programming models can be applied without further improvements. In contrast, heterogeneous tile-based architectures require more sophisticated programming models, especially for accelerator-centric tile-based architectures. Higher compute performance is achieved by tile-based architectures compared to multi-core systems as shown in Figure 2.2. Moreover, shifting from traditional multi-cores to tile-based helped to alleviate interconnect scalability issues which improve the computing performance scalability and increase the overall memory bandwidth, where memory bandwidth is strongly affected by the low scalability of traditional multi-core architectures.

Tile-based architectures typically use a scalable NoC interconnect, which provides more degree of scalability as well as higher memory bandwidth per tile compared to limited shared bus flexibility. The roofline model shown in Figure 2.2 shows a higher memory bandwidth on the diagonal roof which results in a higher compute performance in comparison with multi-core architectures. Therefore, applications with lower operational intensity (memory-intensive) can profit more from tile-based architectures. On the other hand, compute-

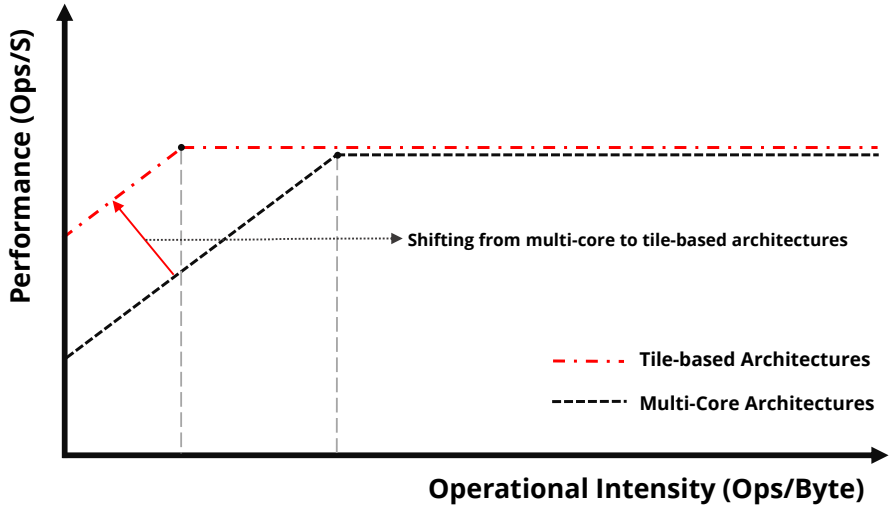


Figure 2.2: Roofline models for baseline multi-core and tile-based architectures [37], [10] showing performance improvement for memory-bound applications running on tile-based architectures.

intensive applications are represented on the horizontal roof line with approximately the same achieving compute performance on both tile-based and traditional multi-core architectures.

Typical tile-based architectures consist of a 2-D grid of heterogeneous compute elements with different memory hierarchies and peripherals tiles. As shown in Figure 2.3, heterogeneous compute tiles can be a cluster of general-purpose cores or a group of domain-specific accelerators. General-purpose tiles are powerful, full-featured computing systems that can host several types of processing cores and independently run an entire operating system. On the other hand, accelerator tiles are specifically assigned for domain-specific computing hosting a broad range of custom hardware accelerators either generated from HLS tools or designed through RTL design flow. DSP, NPU, or DPU compute units can be hosted by accelerator tiles as application-specific accelerators for signal processing and machine learning domains.

Moreover, accelerator tiles usually feature PLMs to increase data locality and accordingly the overall tile computing performance. Several memory technologies are commercially available at the moment (e.g. DDR, HBM, HMC, etc.). Therefore, several tile-based architectures have a set of dedicated memory tiles acting as a shared memory between general-purpose tiles as well as accelerator tiles. Memory tiles can host on-chip or off-chip memory with required memory controllers and data mover units as well as caching levels in case of data coherency requirements. Also, handling data transfer and communication with external peripherals or other external systems require a specific tile architecture for this purpose. Therefore, peripherals or I/O tiles are developed to host required interfaces and communication protocols (e.g. PCIe, Ethernet, UART, etc.) to act as a bridge between compute or accelerator tiles and external peripherals or other computing systems.

Several tile-based architectures are proposed by literature targeting general-purpose and domain-specific workloads with novel tiles architectures for computing, accelerators, and memory. For example, Flex-Tile [35] , GRVI-Phalanex [38] , AsAP [39] , MITRACA [40] , OpenPiton [41] , ESP [13] , and Invasic [42]. In the following subsections, a detailed literature review

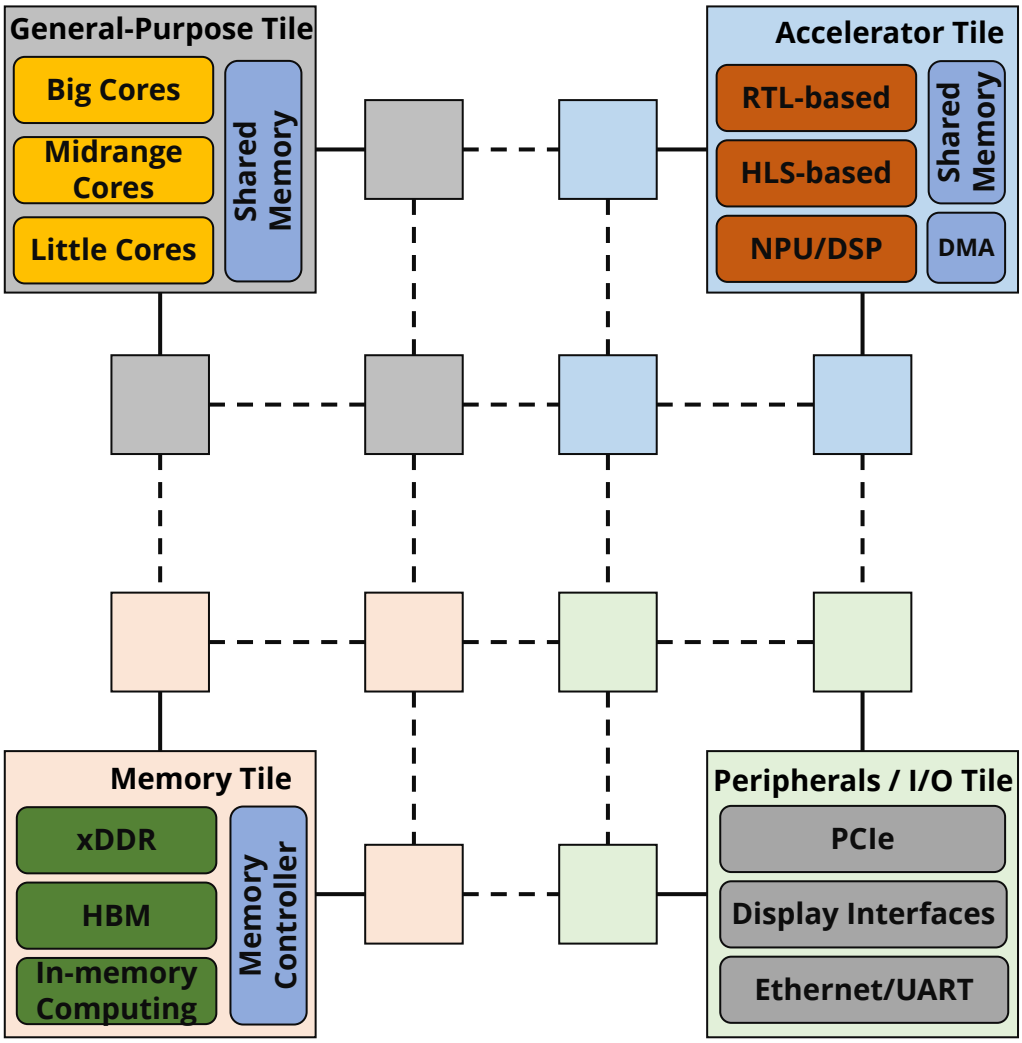


Figure 2.3: Heterogeneous tile-based structure for modern many-core based SoC including general-purpose, accelerators, memory, and peripherals tiles.

of tile-based architectures and their specifications will be presented based on the following points.

- Degree of heterogeneity by supporting multiple ISAs and hardware accelerators
- Supporting open-source ISAs
- Accelerators coupling techniques
- Design modularity and reusability
- Degree of extensibility and configurability

2.1.1 Various Tile-based Platforms

The successful evolution of the very-large-scale of integration (VLSI) technology enabled the development of a large variety of heterogeneous multi- and many-core architectures.

Recently, ARM announced its 5nm Tri-Gear CPU subsystem for mobile SoC [3]. It consists of heterogeneous ISA multi-core systems of several ARM Cortex CPUs. The system provides a balance of power and performance using several core sizes for different workloads. Hence, current trends for the development and implementation of many-core computing systems are to provide heterogeneous computing capabilities with a balance between power and performance targeting a broad spectrum of workloads. Therefore, tile-based platforms can fit as an architectural class suitable to implement heterogeneous many-core systems for a broad spectrum of workloads achieving a balance between power and performance by selecting suitable computing tiles based on workload requirements. Techniques for dynamic adaptivity and voltage frequency scaling are vital to accommodate new many-core architecture classes. In this subsection, various state-of-the-art types and implementations of tile-based many-core architectures are presented and analyzed.

Open-Piton

OpenPiton platform [43] is proposed as an open-source framework to enable the design and development of scalable homogeneous general-purpose many-core architectures. OpenPiton framework provides computing scalability from 1 core to thousand cores [41], supporting several core sizes (i.e. small size CPU: OpenSPARC T1 Core [44], application class CPU: Ariane core [45]). OpenPiton is based on a tiled many-core architecture as shown in Figure 2.4. It consists of two levels of scalable hierarchy. The first level is the chip level, where each chip contains a scalable number of homogeneous compute tiles. Each tile hosts a single CPU with associated levels of caches and NoC routers and interfaces. Within each chip, a coherent multi-plan mesh-based NoC is used for communication and interconnection between tiles. On the upper level of the OpenPiton hierarchy, multiple chips are clustered forming a scalable many-core system. Within a chip, the designer can select the type of tiles to be processing, memory, or I/O tiles. A chip bridge is used to connect the intra-chip NoC to the inter-chip NoC of the second level of the OpenPiton hierarchy.

Data coherency is maintained between different chips for the whole system architecture. By looking deeper inside the chip to figure out the tile architecture and degree of configurability, each tile features the flexibility to host different types of processing cores with configurable

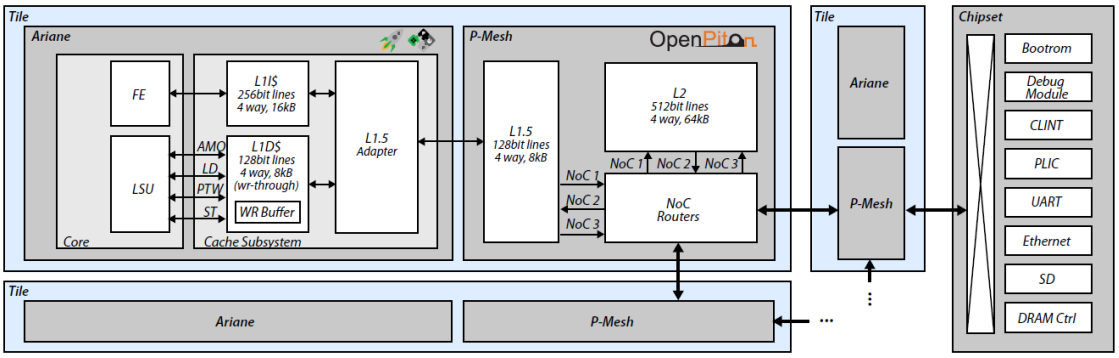


Figure 2.4: Overview of the Open-Piton tile-based architecture [43]. The general-purpose tile contains a single RISC-V core (Ariane, RV64ISA), private caches, and multi-plane NoC routers.

sizes and levels of memory caches subsystem. The cache hierarchy supports up to three cache levels, with private L1 and L1.5 caches (as shown in Figure 2.4 inside the cache subsystem attached to Ariane core) and a shared L2 cache. Furthermore, the used NoC architecture supports data coherency and large data bandwidth by implementing the concept of multi-plane NoC [46] using several physical networks for both inter- and intra-chip hierarchies. On the other hand, OpenPiton features a high degree of interconnection configurability by supporting several coherent interconnection mediums inside the chip. The NoC can easily be replaced by a crossbar or a higher radix design. Other coherent NoC prototypes can easily be integrated to evaluate their effects on the total energy and performance. In terms of portability, OpenPiton was prototyped and ported for multiple FPGA devices through RTL synthesis as well as ASIC design flow. In addition, OpenPiton is extensible by supporting seamless core replacement taking advantage of unified intra-tile interconnection between the core and cache levels. Moreover, AXI interfaces are supported to provide connectivity to a wide range of I/O devices as memory-mapped I/O to the NoCs.

Memphis

Memphis framework [47] is proposed for modeling and generation of many-core SoCs. The framework supports the integration of processor nodes, NoC, and peripherals to models and generates multiple taxonomies of many-core architectures. Memphis supports both SystemC for modeling to speed up simulation time and RTL model for prototyping over FPGA devices. Whereas, the framework integrates both modeling and prototyping into one EDA framework that can be used easily in research and teaching. The framework emphasizes several EDA features to cope with trends of many-core SoC generation including modular logic design flow, automated hardware generation, and debugging methodologies. Apart from EDA features, our focus is on architectural characteristics of the Memphis framework specifically its tile-based architecture.

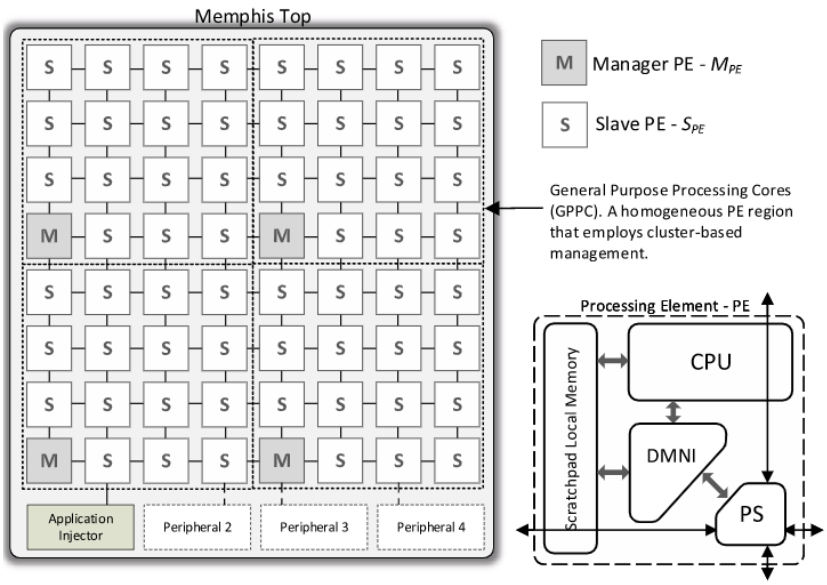


Figure 2.5: Overview of homogeneous tile-based Memphis architecture [47]. Each tile features a single CPU with shared local memory and a NoC router.

Memphis is based on a single level of tile-based hierarchy compared to OpenPiton framework [41], where a 2-D mesh topology NoC is used for communication and interconnection between tiles. As shown in Figure 2.5, Memphis architecture consists of a set of homogeneous processing tiles where each tile hosts a single processing element. The whole 2-D architecture is split into several quadrants where each quadrant has a single manager tile and multiple slave tiles. Both manager and slave tiles feature the same processing element architecture. The processing element hosts a single CPU that could be a MIPS-like architecture (i.e. the Plasma Processor [48]), RISC-V, or ARM. Besides, scratchpad local memories for instruction and data are tightly coupled with the single CPU inside the processing tile.

As Memphis many-core architecture is a NoC-based architecture in terms of interconnection, a direct memory interface (DMNI) is integrated within each processing tile to support simultaneous transmission and receiving of data from local memory to the NoC. The DMNI consists of a network interface and a DMA to connect the NoC router to local memory directly providing a higher memory access rate. Memphis relies on the Hermes 2-D non-coherent NoC [49]. In terms of portability and extensibility, the Memphis framework has been only prototyped targeting FPGA devices with limited extensibility of I/O peripherals through busses within processing tiles.

Black-Parrot

BlackParrot platform [50] is proposed as an open-source RISC-V based many-core platform for heterogeneous acceleration. BlackParrot differs from other many-core platforms by exploiting the openness of RISC-V ISA to build a heterogeneous many-core accelerator. BlackParrot is not dealing with heterogeneous ISAs like other many-core platforms [16], [4].

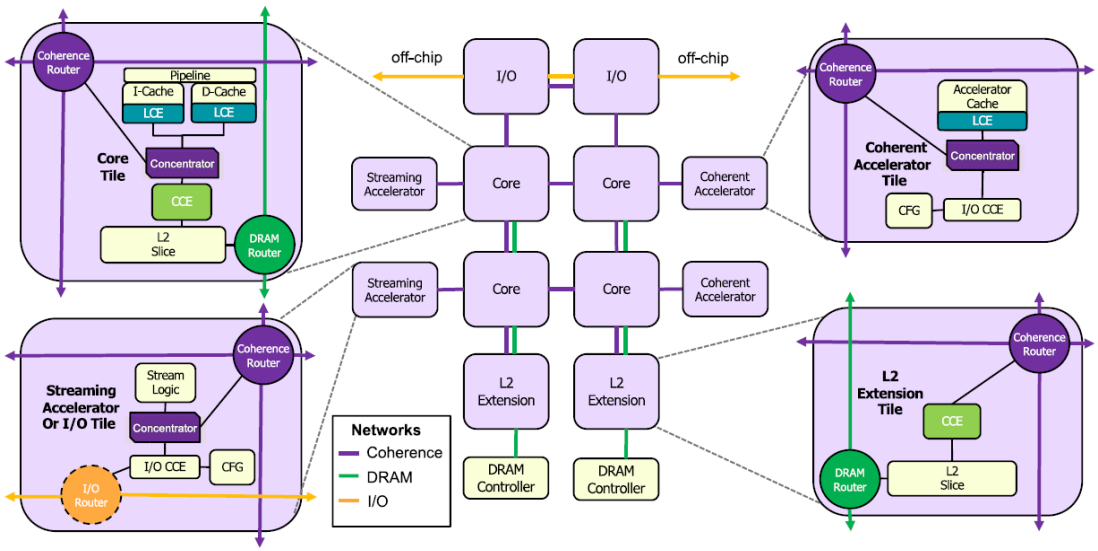


Figure 2.6: Overview of heterogeneous tile-based BlackParrot architectures [50]. It supports three types of heterogeneous tiles: (a) a general-purpose tile with a single RISC-V processing core, (b) a coherent accelerator tile with cache memory, (c) a streaming accelerator tile with a direct connection to external I/O as well as a coherent connection to other tiles.

However, it provides solutions for integrating different sorts of custom hardware accelerators with general-purpose RISC-V cores. Therefore, design modularity and tile-based approach are adopted by BlackParrot platform providing sets of general-purpose, accelerator, and memory tiles as shown in Figure 2.6.

BlackParrot implements a similar interconnection type to the one used by OpenPiton framework [43] based on a coherent multi-plane NoC to support data coherency between all tiles with high data bandwidth. A single level of the tile-based hierarchy is adopted with a single compute element per tile either a custom accelerator or a RISC-V-based processing core. Looking deeper into the general-purpose tile, it supports a single RISC-V core based on 64-bit ISA and Linux-capable. However, the general purpose tile lacks a certain degree of configurability to be adapted with other ISAs. On the other hand, accelerator-based tiles feature a high degree of configurability to host streaming or coherent-based accelerators either generated through HLS tools or by RTL design methodologies.

BlackParrot supports a single type of RISC-V core based on RV64G ISA supporting atomic and floating-point operation with virtual memory to run an operating system. Besides two levels of cache subsystem inside the general-purpose tile. Seamless integration of custom hardware acceleration is the main focus of the design of BlackParrot as it supports coherent, non-coherent, and stream accelerators through modular tile architectures. Also, a memory tile architecture is proposed to control and manage off-chip main memory (e.g. DRAM). In terms of extensibility, BlackParrat features a high degree of extensibility with different workload accelerators with general-purpose compute units. On the other hand, in terms of portability, BlackParrot is only prototyped using ASIC design flow on a 12 nm technology.

P2012

P2012 [51] is proposed as an early ecosystem for a modular and scalable embedded computing accelerator from STMicroelectronics. The primary goal is to achieve high energy efficiency by combining general-purpose computing with domain-specific acceleration realizing an early prototype of a domain-specific architecture. P2012 many-core architecture is implemented based on multiple globally asynchronous locally synchronous (GALS) clusters supporting fine-grained power management. P2012 clusters are connected through an asynchronous global NoC (GANoC) [52]. Each cluster represents a heterogeneous compute tile with a general-purpose multi-core system and loosely coupled hardware accelerators. A local interconnect based on a logarithmic interconnection architecture [53] is used for communication between heterogeneous components within the cluster.

Hardware synchronizers are supported within the cluster to provide scheduling and synchronization for acceleration between the hardware accelerator and general-purpose cores. Besides, a cluster control unit manages data transfer between the NoC and cluster computing subsystem. P2012 provides a modular architectural template to create programmable accelerators by extending it with custom hardware accelerators. On top of the P2012 architecture, a software stack was developed for parallel programming based on OpenCL. P2012 can be attached to a host CPU to act as a programmable accelerator for offloading compute-intensive kernels. P2012 is implemented using STMicroelectronics' low power 28nm CMOS process [54] with the possibility to be interfaced with FPGA devices.

TaPaSCo

TaPaSCo framework [55] is proposed with a similar approach to P2012 [51] as a programmable accelerator attached to a host CPU and supporting a parallel programming method like OpenCL. Moreover, TaPaSCo features a high degree of portability to a broad range of FPGA devices which makes it affordable for research and educational purposes. The main goal of TaPaSCo is to enable an automated design space exploration for FPGA-based acceleration with heterogeneous components. From a domain-specific architecture perspective, TaPaSCo can be considered as a middleware toolflow for domain-specific acceleration providing both hardware architecture layer and software stack targeting FPGA devices. As shown in Figure 2.7, TaPaSCo architecture consists of multiple processing clusters connected through AXI-4 based interconnects for control and data signals. Each processing cluster hosts multiple heterogeneous processing elements that could be a single general-purpose core (i.e. RISC-V, or a Microblaze core) or a custom HLS-based hardware accelerator.

Similar to inter-cluster connection, intra-cluster interconnection is based on AXI-4 interconnects to connect multiple PEs within one cluster. A general-purpose PE features local scratchpad memories for data and instruction to increase data locality for memory-intensive applications. TaPaSCo provides a hardware abstraction layer for seamless integration of a scalable number of general-purpose cores and HLS-based accelerators through an automated tool flow supporting an automatic hardware integration and uniform programming interface. TaPaSCo features a high degree of extensibility by supporting seamless integration of HLS-based accelerators from Xilinx HLS tools with the option to insert private memory between hardware accelerators and FPGA's external DDR memory.

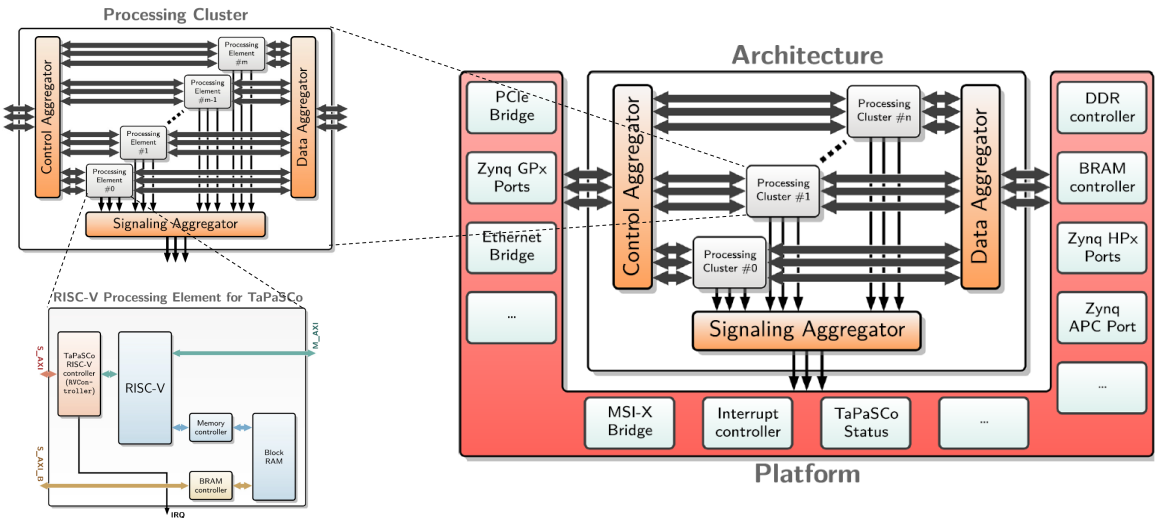


Figure 2.7: Overview of TaPaSCo architecture for parallel reconfigurable computing systems [55]. It consists of multiple heterogeneous processing clusters. Each processing cluster hosts multiple processing elements with a single RISC-V core per each.

MemPool

MemPool [56] is proposed as a shared memory homogeneous tile-based many-core architecture with a special focus to ensure low latency and efficient access to L1 memory among all cores. As shown in Figure 2.8, Mempool architecture consists of two levels of hierarchy. The first hierarchical level is the processing tile where multiple Snitch cores [57] with 16 scratchpad memory banks, each core has a dedicated port to access the memory with one cycle latency. Each tile has a 4-way L1 instruction cache and AXI interconnect is used for communication between memory and cores. MemPool provides the flexibility for all tiles to access the L1 memory of each other. Therefore, extra control units are inserted per tile to handle memory requests and response signals between the tiles. The second hierarchical level is the MemPool cluster where multiple tiles are connected to form a cluster or a group of processing tiles. Several network topologies are used for global interconnections between tiles and between groups of tiles based on logarithmic interconnects.

MemPool has been prototyped using 22nm technology with 256 cores and 1 MiB of shared memory. It achieves a low energy consumption due to memory access optimization mechanisms. However, MemPool is not extensible to support different processing cores or custom hardware accelerators as well as its limited portability to other CMOS technologies and FPGA devices.

ESP

ESP [58] is proposed as an open source research platform for heterogeneous SoC generation. ESP platform features a modular tile-based architecture for general-purpose and domain-specific workloads. It features four types of tile-based architecture for general purpose (processor), memory, accelerator, and I/O tiles. ESP offers an automated solution to integrate custom or third-party hardware accelerators into a complete SoC for what is called agile hardware development [5]. ESP architecture consists of one hierarchical level of heterogeneous tile grid connected through a multi-plane coherent NoC [59] as shown in Figure 2.9. Looking deeper into tiles architectures, the processor tile hosts a single core that is chosen at design time to be either a 64-bit Ariane core [45] or SPARC 32-bit LEON3 core

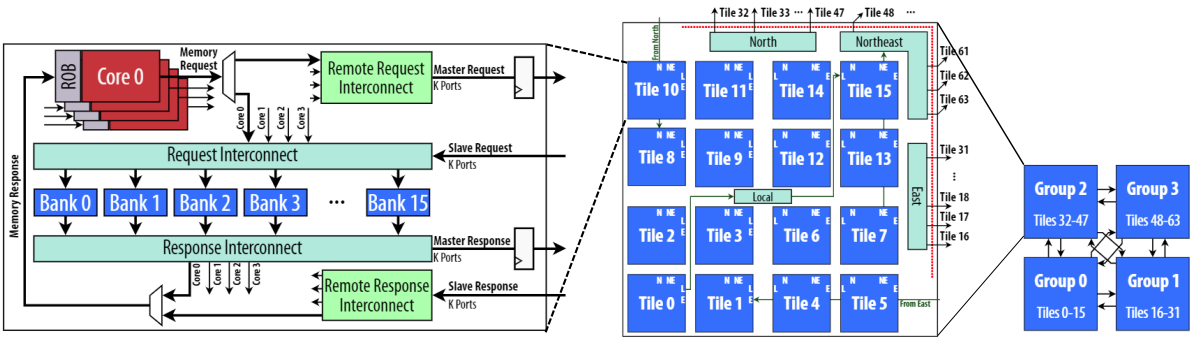


Figure 2.8: Overview of MemPool architecture for general-purpose computing [56]. The architecture consists of multiple clusters, each cluster hosts several general-purpose tiles. Each tile is based on a multi-core RISC-V architecture based on the PULP platform.

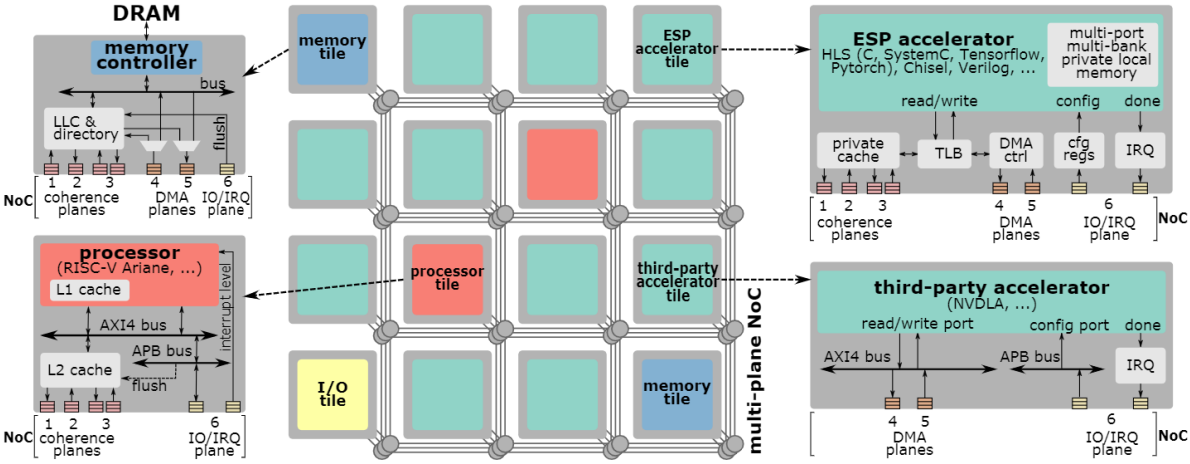


Figure 2.9: Overview of heterogeneous tile-based ESP architecture [58]. ESP consists of four types of tile-based architecture: (a) a general-purpose tile hosting a single core CPU based on RISC-V ISAs, (b) an accelerator tile for HLS-based custom accelerator, (c) an accelerator tile for third-party accelerators (e.g. DSP, NPU), and (d) a memory tile for off-chip memory integration.

from Cobham Gaisler [60]. Processor tiles feature modular architecture with two levels of caches with unified interfaces to support 32-/64-bit operations and memory transactions.

Two accelerator tiles are proposed for loosely coupled integration of HLS-based autogenerated accelerators or third-party accelerators (e.g. NVDLA [58]). Accelerator tiles support load/store ports between accelerator private local memory (PLM) and coherent NoC. Coherent and non-coherent DMA models are supported for several acceleration modes. In addition, the ESP memory tile contains a channel to external DRAM. Several DRAM banks could be supported by several memory tiles. Each memory tile contains a configurable-sized last-level cache (LLC) connected to the NoC plane through an LLC-coherent DMA. Lastly, the ESP auxiliary tile hosts all shared peripherals (e.g. Ethernet, UART, etc.), debugging, and monitoring modules for performance monitoring. ESP provides a full software stack with the accelerator's API library to simplify the invocation of hardware accelerators from a user application making the integration of hardware accelerators as transparent as possible. ESP platform is highly portable to several FPGA devices as well as for ASIC design flow [61].

Manticore

Manticore [62] is proposed as a general-purpose high-performance tile-based many-core architecture for data-parallel floating-point workloads such as data analytics, and scientific computing. Manticore can be classified as an ultra-energy-efficient high-performance computing platform due to its data-path architecture in comparison with baseline GPUs. Also, it supports a certain degree of heterogeneity by supporting both 32- and 64-bit RISC-V ISAs through big and little processing cores. Manticore architecture is based on two levels of computing hierarchy hosting its computing cores as shown in Figure 2.10. The first hierarchical level is the tile or the cluster level which hosts eight Snitch cores [57] based on RV32ISA with

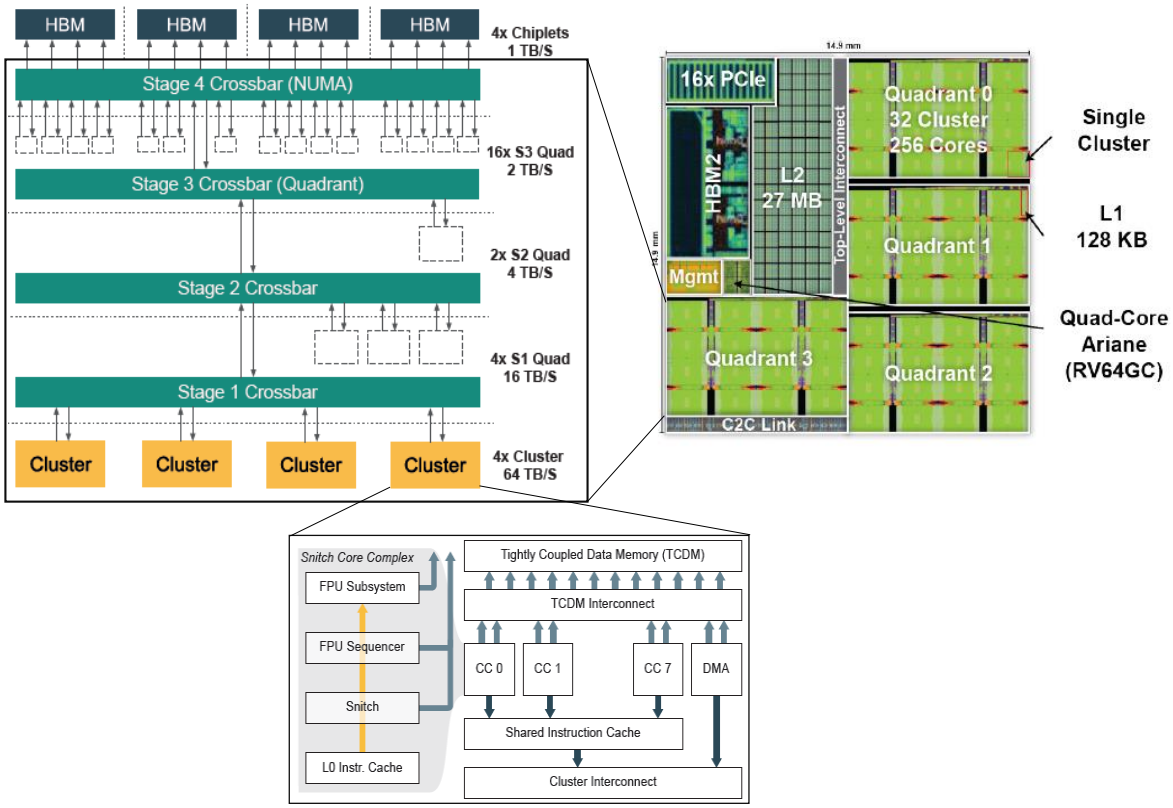


Figure 2.10: Overview of Manticore architecture for general purpose computing [62]. Manticore consists of hundreds of general purpose RISC-V based cores (Snitch core [57]) grouped within multiple processing clusters. The architecture has four large processing quadrants hosting processing clusters and connecting them to HBM.

a single-precision floating point unit. In addition, each cluster has a shared instruction cache that acts as an L2 cache while each core has its private L1 cache.

A tightly coupled data memory is used as a shared memory between Snitch cores connected through a logarithmic interconnect. The cluster is communicated to another cluster through cluster interconnect based on AXI standard, and an internal DMA is used to directly transfer the data between the tightly coupled memory and cluster interconnect. The second level of hierarchy consists of several quadrants where each quadrant hosts several computing clusters. Manticore chip consists of four quadrants, each quadrant hosts in total 256 cores. All computing quadrants are connected through high-performance AXI crossbars [63] in a cascaded method achieving a high data rate between computing clusters.

Manticore is also equipped with a separate processing tile for management based on Ariane core (RV64G ISA). In the last stage of the Manticore hierarchy, four high bandwidth memory (HBMs) are connected for a peak memory bandwidth of 1 TB/s. Manticore has limited extensibility with heterogeneous accelerators while its peak computing performance exceeds baseline CPUs and GPUs by 5x. In terms of portability, Manticore is only prototyped using ASIC design flow with 22nm CMOS technology.

2.1.2 Open-Source RISC-V ISA

In this dissertation, RISC-V ISA has been selected for the implementation of general-purpose processing cores for the tile-based many-core architecture as will be presented and discussed in the following chapters. Therefore, this subsection provides a background and overview of RISC-V ISA and related processing cores microarchitecture. The RISC-V ISA was first introduced at UC Berkeley in 2010 as an open-source ISA for academic and industrial microarchitecture development [6]. RISC-V opens a new wave for new developments and innovation in processor and many-core architectures domains. In comparison with other ISA, RISC-V provides several advantages and new opportunities for microarchitecture developments, especially for academic and educational uses as well as small-size companies that seek fast time to market for their microarchitecture products [64].

The open-source characteristic allows microarchitecture developers to develop and produce their own processing cores from the first design stages till producing chip layouts without prohibitive license costs for non-open-source ISAs. In addition, RISC-V ISA extensions open the door for the development of extensible systems specifically for domain-specific accelerators such as application-specific instruction set processors (ASIPs), co-processors, and tightly coupled accelerators. Modularity is yet another aspect of RISC-V ISA that makes it suitable for a wide spectrum of computing platforms from high-performance to low-power processing cores as well as specialized processors with dedicated execution or accelerator units. Therefore, these advantages led to the proliferation and adoption of RISC-V ISA during the past few years. Nowadays, RISC-V is supported and maintained by RISC-V international organization [65] providing a strong and sustainable RISC-V ecosystem. Over the past few years, several studies have focused on different issues related to RISC-V microarchitecture, security, compiler, and operating system [66], [67].

As shown in Figure 2.11 the number of technical and scientific publications is growing

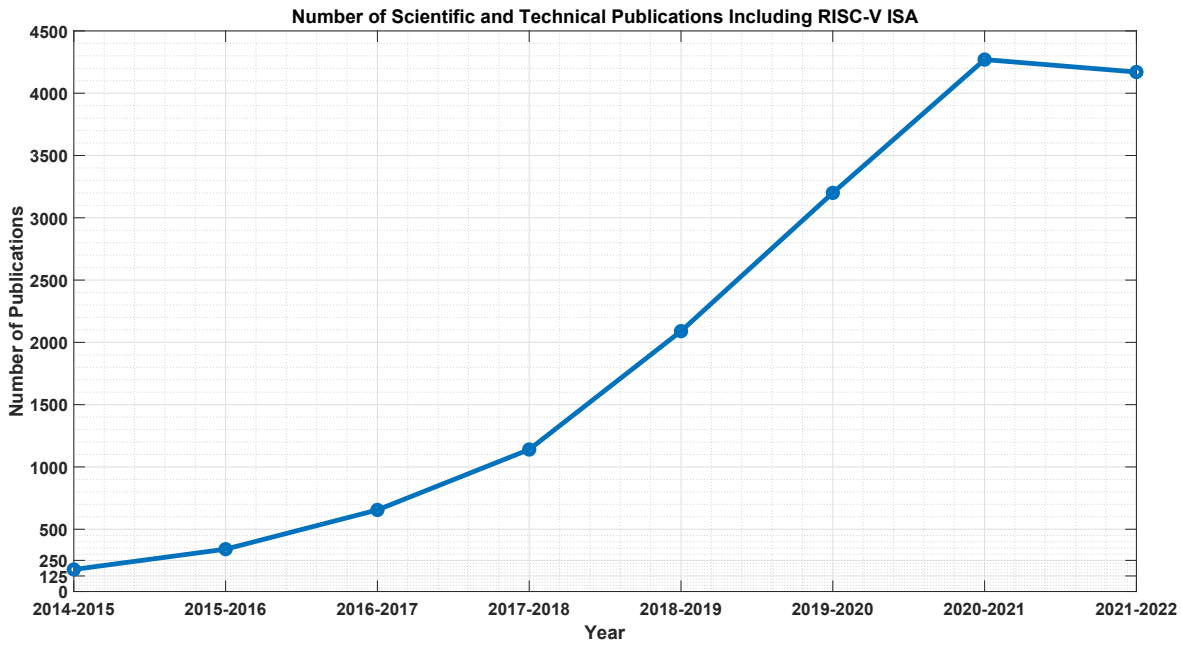


Figure 2.11: Number of RISC-V-based scientific and technical publications since 2014 according to Google Scholar records.

exponentially since 2014 based on google scholar yearly records which reflects the growing interest in the development and use of RISC-V ISA in many domains. As the focus of this PhD thesis is on modular and adaptive many-core architectures, RISC-V-based processors supporting multiple ISAs are used inside several proposed general-purpose tiles [16]. Despite several open-source RISC-V cores being available and ready to use [68], some critical factors are important to be supported by the chosen cores to fulfill the requirements for a modular tile-based many-core architecture. Among those criteria, the processing cores should be developed by a standardized hardware description language (HDL) (e.g. VHDL, Verilog, System Verilog) to be compatible with the rest of the many-core system components.

In addition, RISC-V cores should support external memory subsystems to be extended later with local scratchpad memories to create base PEs for general-purpose tiles. Moreover, selected RISC-V cores should at least support M extension for multiplication and division instructions in order to execute basic arithmetic operations. Factors like area and power-optimized cores are considered during the selection to achieve better utilization and less power consumption on the target FPGA.

Several low-power, embedded class, application class, and high-performance class processors based on RISC-V ISA are developed and implemented both by academic research and industry with several RISC-V extensions and bit widths. Table 2.1 shows current ISA extensions supported by RISC-V and their description. Large varieties of RISC-V cores are currently available supporting and implementing various ISA extensions with different pipeline depths for a broad range of application domains. Table 2.2 shows a list of selected RISC-V cores from low-power application domains to high-performance computing. In other words, from little and midrange cores to high-performance cores. The presented cores in Table 2.2 are classified based on the core size and application domain into three groups. The first group is for little cores or processing cores that can be used for low-power application domains such as wearable and battery-powered devices. In this group, the pipeline structure is short in depth with two to five stages supporting basic arithmetic and load/store operations. Little

Table 2.1: RISC-V ISA extensions [69].

RISC-V ISA Extension	Description
RV32I	Base integer instruction, 32-bit
RV64I	Base integer instruction, 64-bit
RV128I	Base integer instruction, 128-bit
M	Multiplication extension
C	Compresed extension
A	Atomic extension
F	Floating point extension
D	Double-precision floating point extension
G	Supporting M, A, F, D extensions
V	Vector extensions

cores are characterized by small hardware footprints which makes them suitable for large-size many-core architectures to achieve higher performance per watt. However, supporting compute-intensive applications or hosting an operating system is not possible to achieve with single little cores without memory management or virtual memory support.

The second group of RISC-V cores is the midrange core or application class core [87] which supports a deeper pipeline architecture compared to little core microarchitecture. Also, application class cores support floating point extensions for both 32- and 64-bit based on the

Table 2.2: A list of selected RISC-V-based cores.

RISC-V Core (Group)	ISA	Language	Open Source	Pipeline Structure	[ref.]
Orca (Little-core)	RV32IM	VHDL	✓	3-stages	[70]
PicoRV32 (Little-core)	RV32IMC	Verilog	✓	1-stage	[71]
Taiga (Little-core)	RV32IMA	SystemVerilog	✓	3- or 4-stages	[72]
VexRiscv (Little-core)	RV32IMCA	SpinalHDL	✓	2- to 5-stages	[73]
Ibex (Little-core)	RV32IMC	SystemVerilog	✓	2-stages	[74]
Shakti E class (Little-core)	RV64/32IMAC	Verilog	✓	3-stages	[75]
RI5CY (CV32E40P) (Little-core)	RV32IMFC	SystemVerilog	✓	4-stages	[76], [77]
Ariane (CVA6) (Midrange-core)	RV64GC	SystemVerilog	✓	6-stages	[78], [45]
NOEL-V (Midrange-core)	RV64/32IMAFDBCH	VHDL	✓	7-stages	[79]
Rocket (Midrange-core)	RV32/64IMAFDC	Chisel	✓	5-stages	[80]
KLessydra-T (Midrange-core)	RV32IMAV	SystemVerilog	✓	4-stages+ (Vector lanes)	[81]
BOOM (HP-core)	RV64IMAFDC	Chisel	✓	10-stages	[82]
H50XF (HP-core)	RV64IMFDC	Verilog	✗	5-stages	[83]
U7 (HP-core)	RV64GC	Verilog	✗	8-stages	[84]
NX25F (HP-core)	RV64GC	Verilog	✗	5-stages	[85]
XuanTie C910 (HP-core)	RV64GCV	Verilog	✗	12-stages	[86]

base ISA. Application class core as mentioned in Table 2.2 supports a UNIX-based operating system with memory management unit implementations and atomic (A) extension which increase the pipeline complexity and therefore the total hardware footprint. In addition, a virtual address space requires hardware support for fast address translation with transaction lookaside buffer (TLB) and page table to the core. Application class cores are typically connected to off-chip memory. Therefore, the efficiency of memory access relies on the implemented memory hierarchy with caching subsystems. Moreover, the operating frequency for application class cores is much higher than for little cores. Accordingly, computing performance and power consumption are much higher by orders of magnitude which makes them suitable for general-purpose workloads.

The third group includes high-performance cores which usually have deeper pipeline structures than application class cores with single and double-precision floating points extensions. They are also superscalar out-of-order cores with enhanced branch prediction implementations and a complex load/store pipeline stage with several queues [82]. In addition, a distributed scheduler unit is available to support out-of-order execution as well as an enhanced decoding stage. Therefore, high-performance cores are characterized by high computing performance and large hardware footprint which make them suitable for exascale many-core architectures. Exascale many-core architectures are out of the scope of this PhD thesis. Therefore, the selection of RISC-V cores is based on little and midrange cores groups. Accordingly, in the next chapters, the RIC5Y (CV32E40P) [77] and Ariane (CVA6) [45] cores are selected to be used in the proposed multi-ISA many-core architecture as HPC is not supported by the proposed many-core platform.

2.2 Hardware Accelerators Integration

Heterogeneous many-core architectures are increasingly supporting different types of hardware accelerators to achieve the strong need for high computing performance and energy-efficient execution for different application domains. Hardware acceleration provides superior energy efficiency through specialized processing components that can be integrated with general-purpose cores for different workloads requirements. Figure 2.12 shows a SoC architecture that integrates many hardware accelerators with a general-purpose host processor. In this heterogeneous computing paradigm, hardware accelerators are designed to execute specific compute kernels/functions and the host processor runs the remaining kernels. In other words, hardware accelerators are used to offload the execution of compute-intensive kernels from the processor to increase the whole system's efficiency. Hardware accelerators vary in type, flexibility, and efficiency.

Compute-intensive applications require specialized hardware components to improve the performance per watt of selected computational kernels. Some accelerators are highly customized to execute a particular application efficiently such as neural networks [88], [89], [90], THz radar signal processing [91], [92], [93], or computer vision application [94], [95], [96]. Specialized processing can take several ways of hardware acceleration through different design methodologies and architecture types. However, no standard definition for hardware accelerators is available. There exists a large variance for hardware accelerators that differ between them based on accelerator models, degree of granularity, programmability, and way of coupling with the rest of the system.

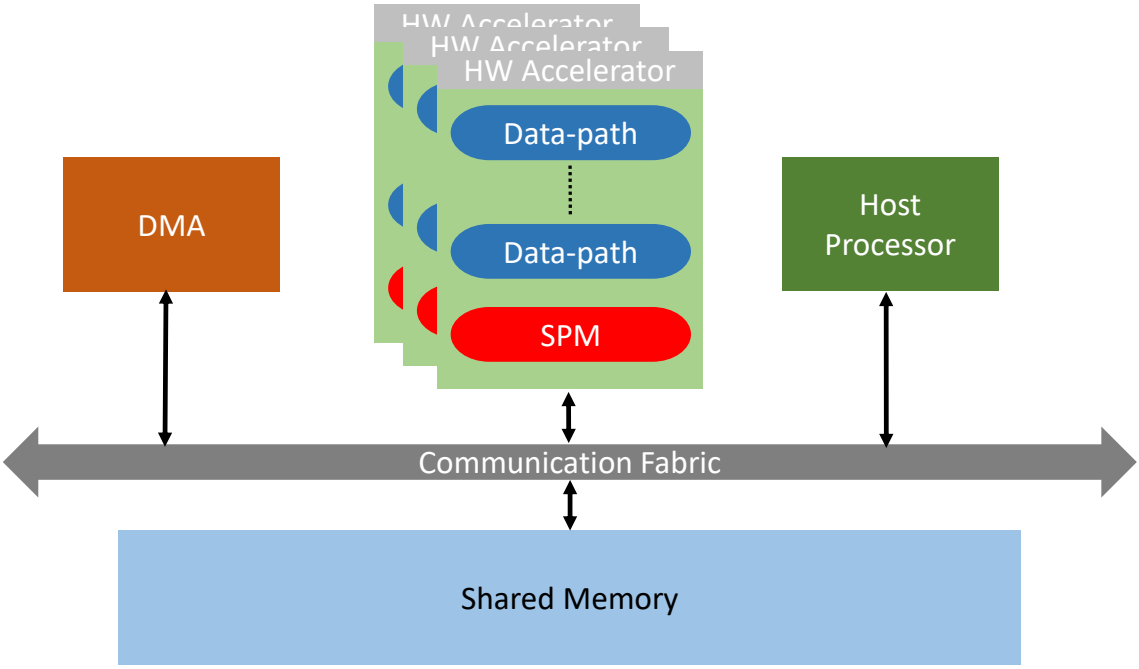


Figure 2.12: Heterogeneous SoC architecture model with many accelerators and a host processor. (DMA: Direct Memory Access, SPM: Scratchpad Memory)

Figure 2.13 shows a categorization of hardware accelerator types.

Fixed function accelerators: a fixed function accelerator is implemented to execute a specific function from an application workload with different configurations and I/O sizes. Prior to fixed function accelerator design, application kernels identification is conducted to determine the compute intensity degree and regularity usage of the target application's functions/kernels in order to specify which functions are efficiently executed using fixed function accelerators. Fixed function implementations provide the highest performance per watt as they are highly optimized for specific operations. However, their efficiency starts to decrease when they are designed to support multiple functions or to be configurable for large design spaces.

Design constraints (e.g. performance, energy, area) are very crucial to determine the specification and number of fixed function accelerators for a specific application domain. For example, in the autonomous driving domain, the conducted study by [97] shows that deep neural network inference and feature extraction consume 95% of the total execution cycle. Accordingly, using fixed function accelerators on an FPGA reduces the overall execution time by 93x compared to software-based implementation. Similarly, for THz synthetic aperture radar, FFT and backprojection algorithms consume 90% of the overall execution time needed to construct one image [91]. Therefore, fixed accelerator units for FFT and projection are implemented and integrated with a host processor using a HW/SW co-design design flow to achieve a speedup latency of 36x compared to software-based implementation. Nevertheless, the use of fixed-function accelerators is limited to one application with a specific set of functions. Therefore, this type of acceleration lacks high level of granularity and flexibility to be reused by different workloads.

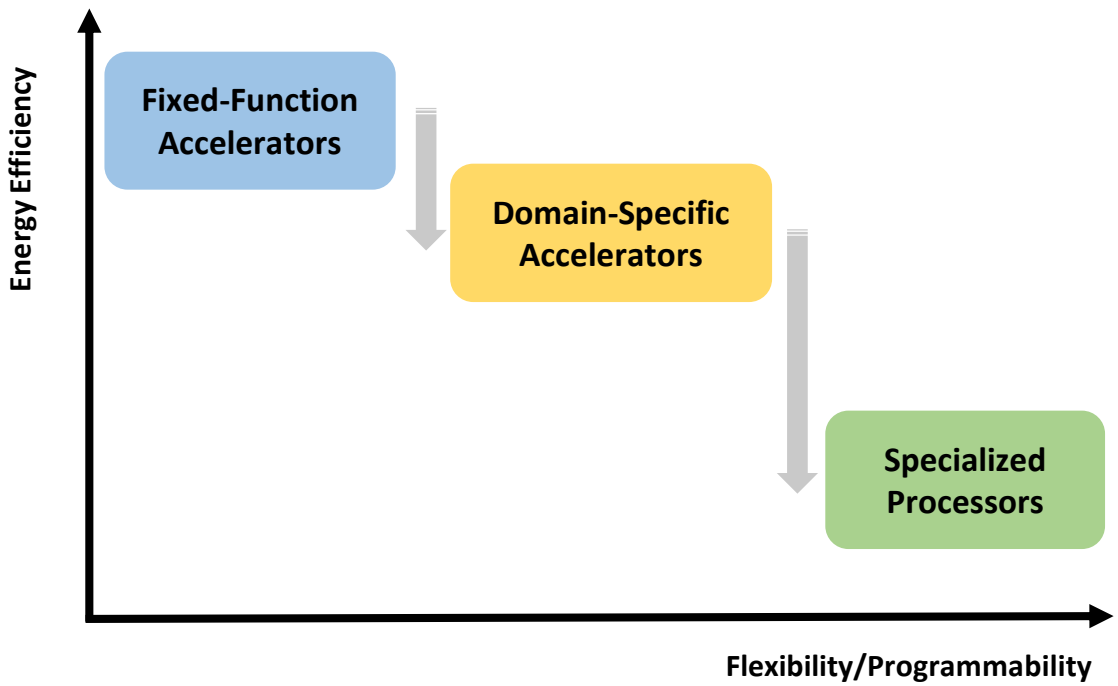


Figure 2.13: Hardware accelerators categories and the related trade-off between flexibility and energy efficiency.

Domain-specific accelerators: domain-specific accelerators are proposed as a solution to increase the granularity and flexibility of hardware acceleration to support and be reused by multiple functions from different workload applications. In general, domain-specific designs provide the flexibility to adapt the same architecture to a set of applications from the same domain [98] compared to general-purpose architectures. This category of acceleration offers a balance between specialization and generality. It is not based on special instructions like will be shown next but it is based on special engines for a domain of algorithms (e.g. matrix multiplication, FFT kernels, etc.).

An early version of a domain-specific accelerator is the function level processor [99] which aims to reduce the gap between flexibility and efficiency. It supports function-level processing instead of traditional instruction-level processing in normal processors. The datapath is a pipeline of functional blocks that are used in a similar way to baseline pipeline stages. Each functional block represents a hardware-accelerated function. For sparse matrix multiplication domain, a streaming domain-specific accelerator is proposed by [100] for a wide range of sparse multiplication techniques. It consists of a two-dimensional array of parametrized MAC units with tightly coupled on-chip memory. The accelerator is configurable at runtime to support different sizes of matrices and multiple sparsity algorithms. Accordingly, domain-specific accelerator increases the degree of granularity and modularity for hardware acceleration that facilitate the integration of a large number of accelerated function in a heterogeneous many-core architecture.

Specialized processors: Specialized processors differ from domain-specific accelerators that they offer a certain level of programmability for more flexibility to support multiple application domains. However, increasing the flexibility will degrade the efficiency by a certain factor. On the other hand, specialized processors reduce the design effort by increasing the design modularity which allows the design reuse into several many-core architectures. In other words, they are seamlessly integrated into different system architectures. Specialized processors can be application-specific instruction set processors, tightly coupled co-processors or custom accelerators with a host CPU. Data-level parallelism and custom instructions are employed in all of these computing paradigms. For example, machine learning domains are highly benefited from specialized processors to achieve higher orders of magnitude higher performance than baseline general purpose computing, such as [101], [102]. In general, the development and design of specialized processors consider a range of kernels or a set of functions to be supported by custom instructions to be invoked from the software layer.

Hardware accelerator efficiency does not only rely on the accelerator architecture type, internal compute units optimization, and degree of parallelism. The coupling and the way hardware accelerators are integrated into the whole system is a crucial design decision that affects the actual performance of hardware accelerators while interacting with other compute and storage units within the system [103]. In this subsection, state-of-the-art techniques for accelerator coupling to processor and memory units are presented.

2.2.1 Accelerator Coupling Models

Accelerator coupling is one of the distinguishing features of a hardware accelerator, as it impacts fundamental design choices for SoC or many-core architectures. Coupling models determine the interaction between hardware accelerators and processors, whether the target accelerator is a memory-mapped peripheral within the address space of a processor or is considered an extension to the processor execution unit stage. Also, memory access patterns between accelerators and local or external memory units are highly related to coupling models. In addition, how accelerators are operated and controlled, through software, or software and hardware; and whether custom instructions are required for the interaction between processors and accelerators. Accelerator coupling models can be categorized into two classes: 1) tightly coupled accelerator (TCA), and 2) loosely coupled accelerator (LCA).

TCA model: In the TCA model, the hardware accelerator is coupled to a general-purpose processing core as an extension to the core itself through ISA extension or as a separate accelerator directly coupled to the processor with/without sharing the data cache memory as shown in Figure 2.14. TCAs are typically specialized data paths that fit with domain-specific and specialized processor acceleration categories. They are tightly integrated compute units with the processor functional unit to offload frequently occurring kernels or operations of an application code. In other words, the execution stage of a processor pipeline is composed of several functional units including TCAs. A tightly coupled model imposes several challenges on the accelerator's design and implementation. The accelerator area should be approximately the same size as other processor pipelines' functional units. The use of local memory in the design of TCAs is unlikely supported to reduce the size of storage elements (i.e. SRAM) in pipeline layouts. Therefore, a limited amount of storage units are implemented as registers and buffers within TCAs.

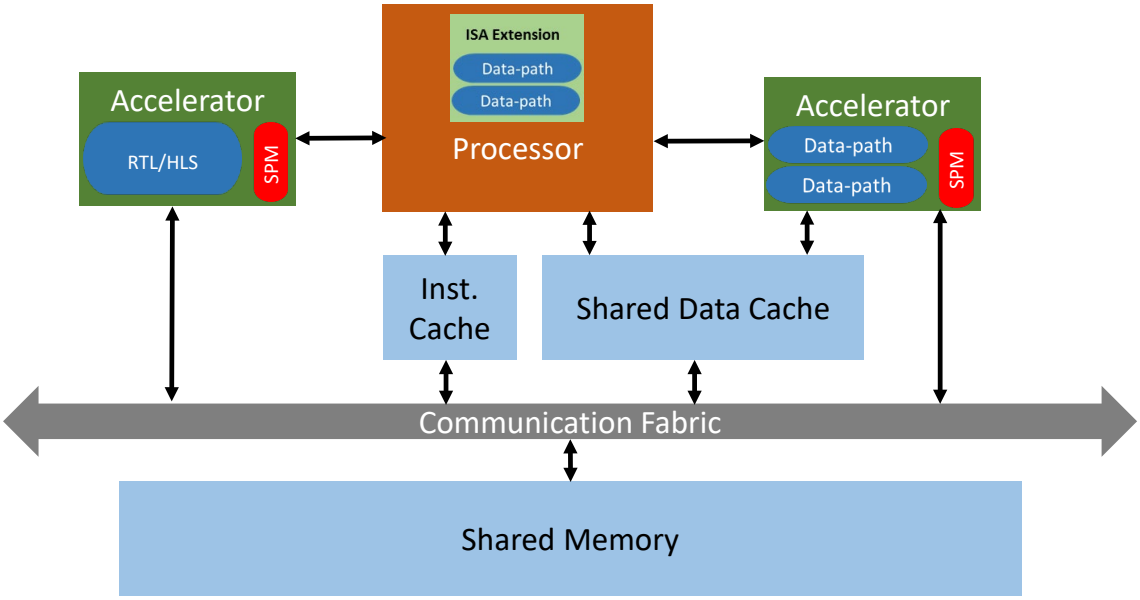


Figure 2.14: Tightly-coupled accelerator model, where accelerators are integrated as an extension to a general-purpose processor or as an accelerator directly coupled to the processor with/without data cache sharing.

Designing TCAs requires ISA extension with special instructions to be diffused through the software via low-level libraries or the compiler. TCAs allow the extension of microprocessors via vector instruction to support single-instruction multiple-data (SIMD). The Tensilica extensible processor [104] is an example of a commercial IP for ISA extension that supports the integration of TCAs to a baseline processor pipeline. On the other hand, open-source RISC-V ISA extension offers the flexibility to integrate custom TCAs into RISC-V pipeline [105]. In this context, RV-CNN [106] is proposed to extend the RV32ISA with custom instruction to accelerate several CNN operations through embedded TCAs to the RISC-V pipeline. Similarly, a vector extension unit is proposed by [107] to support multiple matrix operations acceleration through multiple execution lanes. However, ISA is typically fixed and proprietary which limited the design choices for internal compute units and control path of extended execution units. Co-processors are the second type of TCAs that can perform more complex tasks than a single custom instruction and can handle large data sets. Co-processors can support large sizes of private local memory that could be coherent or non-coherent with the system resources. In this case, accelerators are implemented as separate entities, not authentic parts of the processor, that are integrated with the processor core through dedicated interfaces and interconnections as shown in Figure 2.14 by the accelerator on the left.

Data access through co-processors can support coherency by sharing a private cache with the processor as the example of the right accelerator in Figure 2.14. For cache coherent data transactions, co-processors must implement the same coherence protocol supported by processor cores [108]. Several co-processors implementations support both coherent and non-coherent data transactions depending on target applications' requirements. The CNNX [109] is implemented as a co-processor to accelerate neural network kernels for embedded computer vision applications. It supports multiple neural network topologies and depth as well as different sets of parameters. Similarly, RedMU1E [110] is implemented as a

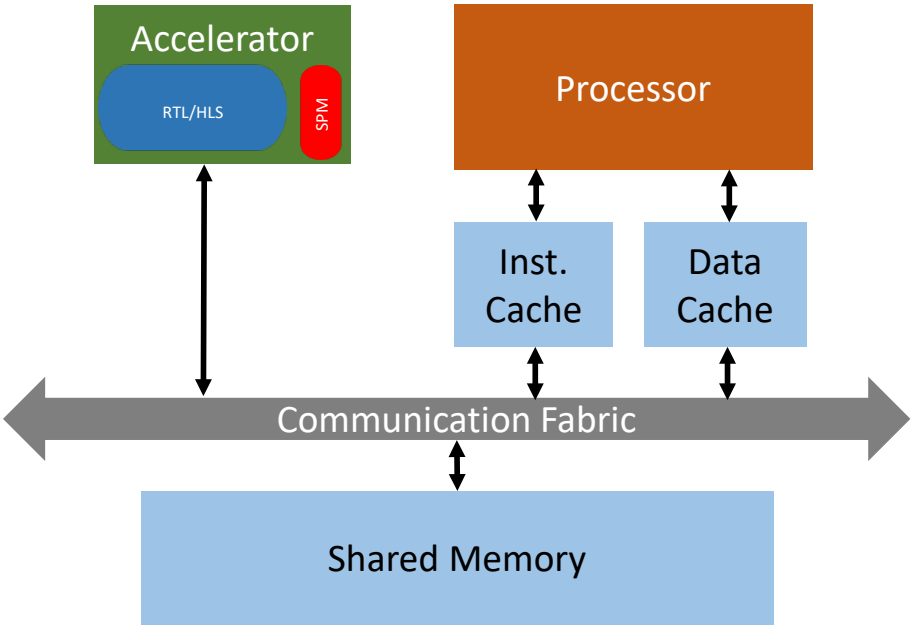


Figure 2.15: Loosely-coupled accelerator model, where accelerators are integrated into the system through a communication fabric as memory-mapped peripherals to general-purpose processors.

tightly-coupled co-processor for matrix multiplication acceleration. It supports floating and fixed point multiplications for deep learning inference. The co-processor communicates to multi-RISC-V-based cores through a shared bus with a shared data memory.

LCA model: In the LCA model, the hardware accelerator is located outside the processor core and interacts with it through an on-chip interconnect as shown in Figure 2.15. As a consequence, LCAs can support larger accelerator sizes with more private memory compared to TCAs. This allows coarse-grain accelerators with complex data paths and large storage units that are capable to accelerate a complete application instead of small kernels or specific functions. Therefore, LCAs feature a high level of parallelism with parallel and multiple data paths. LCAs do not require ISA extensions, they are running independently from general-purpose cores. Instead, they are configured with low-level drivers or libraries similar to memory-mapped peripherals in the system. Moreover, LCAs provide more flexibility by freeing general-purpose cores to run other tasks in parallel with application acceleration.

Typical LCAs are integrated with DMA for direct interaction with storage units without interfering with processor-memory access. Unlike TCAs which are sharing the memory with general-purpose core which degrades their memory access bandwidth. Research works related to LCAs show a long list of different structures of LCAs from different application domains. Hence, LCAs are not limited by the ISA or processor interface protocols like in the case of TCAs. LCAs can be designed independently and decoupled from the system, the only requirement is to support standard input and output interfaces that are compatible with the communication fabric protocol of the target system. CHARM [111] is an early example of accelerator-rich architectures [112] which is based on a massive number of LCAs implementing different computational kernels at different degrees of heterogeneity and granularity. CHARM provides the hardware and software infrastructure to realize a massive heterogeneous accelerator-centric platform for a broad range of application domains.

In the same context, AXR-CMP [113] offers a management scheme for accelerator-rich architectures to support resource sharing between multiple general-purpose cores and LCAs. It allows the creation of virtual accelerators out of multiple smaller ones using a chain of multiple accelerators together. For RISC-V-based SoC, a framework to simplify the deployment of LCAs in a heterogeneous multi-core SoC is proposed by [114]. It supports the seamless integration of HLS-based accelerators by automating the generation of coupling interfaces between the generated accelerator overlay and the system interconnect. Some examples of accelerator-rich architectures with LCAs are the brain-inspired computer MasterMind [115] and the KACHEL platform for 5G signal processing [116]. Both platforms feature a massive amount of domain-specific and fixed function accelerators that can accelerate a whole compute-intensive application either from the machine learning or 5G domains. In addition, they are equipped with sophisticated resource management units to highly optimize resource utilization and increase computing performance at runtime.

Table 2.3 summarizes the main differents between TCA and LCA models regarding the area, memory sizes, supported data sizes, and controlling mechanisms. In addition, LCAs have another main advantage over TCAs as they ease the development and integration in heterogeneous many-core systems. The designer needs to adhere to the same interfaces and communication protocols imposed by the main communication fabric of the target many-core system. During this thesis, the focus of the work is on the LCA model and how to seamlessly integrate this model into the proposed tile-based many-core architecture.

2.2.2 Memory Management for Accelerators

According to the accelerator store framework published in 2010 [118], an average of 69% of hardware accelerator area is consumed by private local memory. In this survey, target hardware accelerators are LCAs integrated into multi- and many-core systems with general-purpose cores. An example of a heterogeneous many-core architecture with multi LCAs with large private local memory integrated with general-purpose compute units [117] is depicted in Figure 2.16. Therefore, addressing memory aspects of LCAs is a necessary step to design an efficient heterogeneous many-core system. Also, supporting data coherency between LCAs' private local memory and shared main memory in the system is another challenge to be addressed. Accordingly, the effect of multiple accelerators processing a large amount of data through off-chip memory needs to be analyzed and considered specifically in cases of memory-intensive applications. A typical LCA consists of several computation units or

Table 2.3: Comparison between hardware accelerator coupling models.

Accelerator Coupling Model	Area & Resources Utilization	Private Local Memory Size	Supported Data Size	Controlling	Examples [Ref.]
Tightly Coupled Accelerators	Small	Small	Small	SW	[106], [107] [104], [109]
Loosely Coupled Accelerators	Large	Large	Large	HW/SW	[111], [113] [115], [116]

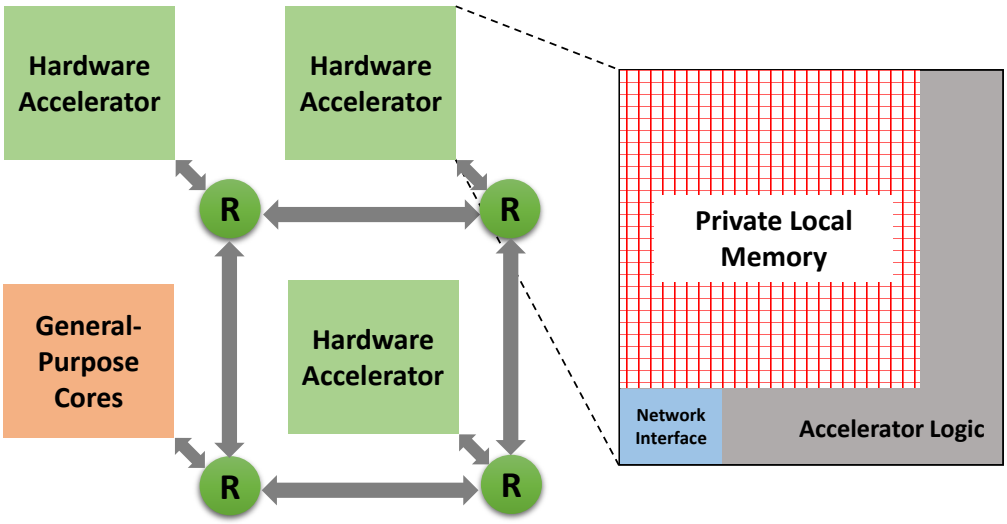


Figure 2.16: An example of a heterogeneous many-core architecture with many LCAs and general-purpose cores, it shows the large size of private local memory that dominates the area of LCAs [117].

accelerator logic that implements the arithmetic operations of accelerated functions and storage units or private local memory (PLM) that stores data as shown in Figure 2.17. PLM constitutes the accelerator memory subsystem and it can be a scratchpad memory in a single or multi-banks or cache memory unit. PLM units are used to store an amount of data from the DRAM in order to handle large data sets workloads. In fact, PLMs can reach up to 90% of the LCA area. However, the amount of data that can be stored on-chip is limited to a few MBs. In this subsection, several accelerator memory interaction techniques and supported features are presented and analyzed based on: 1) supporting direct-memory access and coherency, 2) supporting large data sets, and 3) supporting accelerator memory reuse. Several accelerator memory interaction techniques and supported features are presented and analyzed based on: 1) supporting direct-memory access and coherency, 2) supporting large data sets, and 3) supporting accelerator memory reuse.

Memory Access and Coherency: Several techniques are proposed by literature trying to reduce the communication overhead in private accelerators by optimizing data transfer between memory and accelerator logic. In this subsection, traditional memory technologies (e.g. SRAM, DRAM) are considered for accelerators' local memory and the complete system shared memory implementation. In shared memory accelerator-centric architectures, accelerator sharing takes place at different levels of the memory hierarchy, from PLM to shared main memory. LCAs can be shared at different memory levels (e.g. L1, L2, or last level main memory), it depends on the degree of scalability and hierarchical design of the target many-core system. In this context, a hardware accelerator wrapper is proposed by [120] supporting data streaming between the accelerator hardware logic and the shared tightly coupled data memory of the system. The wrapper hosts either an RTL or HLS-based accelerator with a control path and wrapper interconnect modules. The wrapper implements the control plane for the hosted hardware accelerator. The hardware accelerator wrapper includes a register file that is accessible by processors directly to read and write control signals and

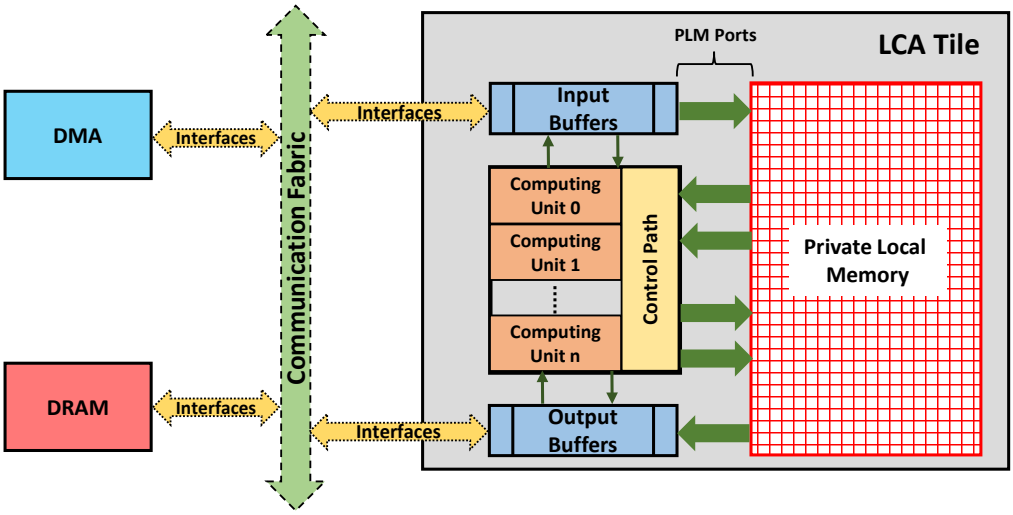


Figure 2.17: LCA tile structure as described by [119]: it contains several computation units representing the accelerator logic, private local memory, control path, I/O buffers, and interconnection interfaces for data transfer.

the status of the hosted accelerator. A synchronization module is implemented to handle data transfer between the accelerator PLMs and tightly coupled data memory of the system through several DMA units. In addition, it supports data moving between multiple hardware wrappers in the system. The process of controlling the hardware wrapper is handled through a set of low-level drivers from processor cores that can be called directly from the software. Overall, the proposed hardware wrapper by [120] allows a smooth integration of accelerators into shared memory many-core systems with SW APIs to facilitate the interaction between accelerators and shared memory. Similarly, Bellochi et al. [121] proposed a RISC-V-based overlay with multiple LCAs. The overlay architecture consists of a multicore RISC-V with shared tightly-coupled data memory connected through a bus interconnect. LCAs are attached to the bus interconnect within a hardware accelerator wrapper that controls data movement between the bus interconnect and the accelerator logic. However, the proposed overlay lacks modularity and has limited scalability. It provides a single method to integrate the accelerator logic through a shared interconnect to a multicore RISC-V without direct access to external memory.

The template-based memory access engine (MAE) [122] is a similar approach proposed to address decreasing memory latency by simultaneous memory access from multiple accelerators in many-core systems. MAE provides a common memory access template for accelerators that can handle different memory access patterns such as streaming, strided, complex, indirect array access, and gather access patterns. It consists of a template-based prefetcher located next to the memory controller to prefetch data from memory to accelerator PLM. MAE aims to reduce memory access latency and jitter in many-accelerator-based SoC. MAE internal architecture consists of a template-based prefetcher and a prefetch buffer. The prefetcher contains a prefetcher table to store memory access request data to predict memory congestion. The scheduler and prefetch request handler are the main prefetcher units to generate memory commands for the memory controller unit. The prefetch buffer

consists of SRAM storage to store received read and write requests to be served by the prefetcher. The MAE is similar to DMA, it provides support for more memory access patterns, not only streaming, for different kinds of accelerators' memory access patterns. Different cache coherence models for accelerators are proposed by literature such as fully-coherent, last-level-cache (LLC) coherent, and non-coherent [123]. The non-coherent model allows the accelerator to access off-chip memory directly. While in the fully-coherent model, accelerators are coherent with private caches of other compute units in the system (i.e. processors, and accelerators). In this context, an extension of a directory-based cache-coherence protocol is proposed by [124] to support coherent LCA in many-core NoC-based architectures. An extension of the MESI directory-based protocol is implemented and integrated into coherent LCAs.

Coherent LCAs are communicated through a coherent multi-plane NoC that supports data coherency between heterogeneous components. The evaluation results show a significant reduction in the number of memory access compared to non-coherent-based accelerators over NoC. In order to manage several cache coherency models at runtime, a runtime reconfigurable memory hierarchy for scalable SoCs is proposed by [125]. It aims to support the previous three listed cache-coherence models by proposing a runtime adaptive algorithm to manage the coherence of LCAs. The proposed algorithm is running on coherent NoC-based architecture with heterogeneous LCA tiles and processor tiles. Cache controllers are implemented inside the LCA tile as a socket between the network interface and the accelerator's PLM. Based on the accelerator memory patterns, the proposed runtime algorithm selects the optimum cache-coherent model. As a result, a 30% reduction in memory access is achieved compared to fixed cache-coherent model implementations.

Similarly, Cohmeleon [108] is proposed to manage multiple cache coherent models for coherent LCAs in heterogeneous SoCs. Cohmeleon applies reinforcement learning algorithms to select the optimum coherence model dynamically at runtime by observing the system and monitoring its performance. It supports different SoC architectures either with NoC or bus-based interconnections. Cohmeleon trains a reinforcement learning model to select the optimum cache coherence model for each LCA in the system. The training model takes into account LCA execution time and off-chip memory access number and patterns of each LCA. As a result, Cohmeleon reduces the off-chip memory access by 66% compared to state-of-the-art fixed coherent model solutions.

Managing Large Data Size: Big data applications with large data sets are increasingly used for data analytics and scientific computing. Typically, large data centers are used to execute such kind of applications with large data sets. In order to reduce energy efficiency and increase computing performance, hardware acceleration could be a suitable solution within a scalable heterogeneous computing system [126]. However, the size on-chip PLM is smaller than the size of data sets. Therefore, handling large data sets by hardware accelerators require new solutions to handle data movement between external memory and accelerators' PLMs. Therefore, an accelerator structure solution is proposed by [119] to handle large data sets for high-performance embedded applications. The proposed accelerator structure supports a direct sharing of physical memory across processors and accelerators with an accelerator virtual address space that is separated from the processor virtual address space.

Moreover, the main feature of the accelerator structure is a dedicated DMA controller with a specialized translation look-aside buffer (TLB) that supports multiple specific memory access patterns. The accelerator contains circular and ping-pong data buffers to support

the pipelining of computation and DMA transfers with off-chip memory. The main focus is to parallelize the computation process with the memory and data movement process and hide the DMA latency in order to not degrade the acceleration performance. The proposed acceleration solution is integrated into an automated toolflow for accelerator memory design called MNEMOSYNE [127]. The toolflow supports an automatic generation and optimization of memory hierarchy for HLS-based accelerators. It optimizes the placement of accelerators with respect to the location of DDR controllers and load-balancing policies. Moreover, it supports a scalable number of concurrent accelerators with different data set sizes.

Accelerator Memory Reuse: The number of hardware accelerators is growing rapidly in recent heterogeneous many-core architecture. Therefore, the required on-chip memory size increases to implement the required accelerators' PLMs and internal buffers. However, hardware accelerators are not utilized 100% during operating time. As a consequence, accelerators' PLMs are remaining unused during the accelerator's inactive time. Therefore, enabling the re-utilization or sharing of accelerators PLMs with other system components can improve resource utilization and increase efficiency [117]. As a result, a drastic decrease in accelerators' cost of integration into heterogeneous systems can be achieved.

In this context, several works have been proposed to reuse accelerators PLMs as a cache memory in a non-uniform cache architecture with the rest of the system. All considered accelerators are LCAs with complex and large data paths with a few megabytes of on-chip memory. BiN [128] is proposed to share accelerator internal buffers in non-uniform access memory many-accelerator systems or accelerator-rich architectures. A highly efficient on-chip utilization has been achieved through several methods to dynamically allocate accelerator buffers in a non-uniform memory access architecture. First, a dynamic interval-based global allocation method is proposed to assign extra free buffer spaces from some accelerators to other accelerators that can best utilize them. In this case, buffers are allocated on demand as an extended cache memory of the accelerator which requests more memory space.

The second proposed method is a flexible and low overhead paged buffer allocation to reduce the effect of buffer fragmentation. In this method, the accelerator will use a small local page table to translate buffer addresses into absolute addresses in order to set the page granularity for each buffer according to the buffer size. Therefore, larger buffers have a larger page size. For BiN, the buffer sizes are set to a limit of a few kilobytes for only accelerator-based architecture which limits its adoption in heterogeneous many-core architectures with general-purpose cores and hardware accelerators. Therefore, ROCA is proposed by [129] to overcome this limitation by sharing accelerators PLMs with any compute element type in heterogeneous many-core architectures. ROCA is using the complete PLM, not just a fraction of it, to extend the cache memory size of another heterogeneous element in the many-core architecture.

In order for accelerator's PLM to operate as a cache block, a cache manager is implemented for each accelerator. Since ROCA extends the many-core system last-level-cache with accelerator PLM. A large tag array in the last-level cache is implemented to track blocks stored in accelerator PLMs. Accordingly, previous works are trying to exploit the abundant PLMs in accelerators to reuse them as extended memory blocks for the whole system while they are not used by accelerated workloads. In this way, a reduction in accelerator integration cost, as well as more energy efficiency can be achieved. In this PhD thesis, an accelerator memory reuse implementation is proposed for FPGA-based heterogeneous many-core architecture. The proposed implementation is based on a hybrid memory/accelerator tile that is described

in Chapter 4, it supports a dual mode of operations to operate as a hardware accelerator with BRAM/URAM-based PLM or using the on-chip PLM blocks as a scratchpad shared memory for general-purpose computing tiles.

2.3 Runtime Adaptive FPGA-based SoC

By the fading of Moore's law and the rising age of heterogeneous computing paradigms and domain-specific architectures [2], agile design practices arise as new topics of research to mitigate the new shift towards highly customized heterogeneous systems. Adaptive or reconfigurable computing is considered one of the agile design practices that can reduce the development cost from the economic point of view [130]. Hence, the development of heterogeneous architectures and associated hardware accelerators is not an easy task, especially in highly scalable systems with a high level of heterogeneity among computing units. Sometimes, the underlying hardware architecture requires to be re-designed and optimized every time a new class or domain of applications needs to be supported which not only increases the development cost but also reduces the ability of upgrading and maintain the system architecture and its associated design toolflows and programming methods. On the other hand, the deployment cost as an ASIC design will increase accordingly due to: 1) complex design tools and required skills, 2) a longer and unsustainable development cycle, and 3) wastage during upgrades or re-design. The main challenge is how to efficiently manage the changes and upgrades without the need to repeat the whole design process. Therefore, adaptive computing is the optimum way to manage the regular changes and upgrades in modern highly scalable heterogeneous architectures [14].

In this section, the focus will be on runtime adaptive FPGA-based systems and their related reconfiguration management frameworks and how adaptability can be used in heterogeneous many-core systems to increase their flexibility and reduces the upgrading cost. An FPGA is a type of integrated circuit design that can be reprogrammed to implement several digital blocks to execute different functions or applications (e.g. digital signal processing, neural network algorithms). FPGAs can be classified into two main types: 1) flash-based and static random access memory (SRAM) based FPGAs, and 2) fuse-based and anti-fuse-based FPGAs. There are several FPGA manufacturers, and two of them are dominating the FPGA market which are AMD (Xilinx) and Intel (Altera) FPGA devices. In this PhD thesis, Xilinx SRAM-based FPGA devices are considered. A Xilinx SRAM-based FPGA has a configuration memory that stores the configuration of the target digital functionality in a form of a bitstream.

A typical Xilinx FPGA device floorplan consists of a grid of resource and interconnects tiles as shown in Figure 2.18. Configurable logic blocks (CLBs) tiles contain a column of CLB where each CLB consists of slices of look-up tables (LUTs), flip-flops (FFs), and multiplexers. LUTs are used to implement logic functions, where they can be combined together via interconnect to form larger logic functions. In addition, Xilinx FPGAs contain on-chip block memory (BRAMs/URAMs) which are also arranged in columns on the same FPGA floorplan. Each memory block can operate independently to perform memory read/write operations in parallel with other on-chip memory blocks. Also, several BRAM/URAM blocks can be combined together to act like a single memory with a continuous address space. BRAMs/URAMs can be used to implement RAM, ROM, and several buffer implementations. A Xilinx FPGA supports simple on-chip digital signal processing units (DSPs) which are also arranged in the form of columns like other resource types as shown in Figure 2.18. The DSP unit consists of

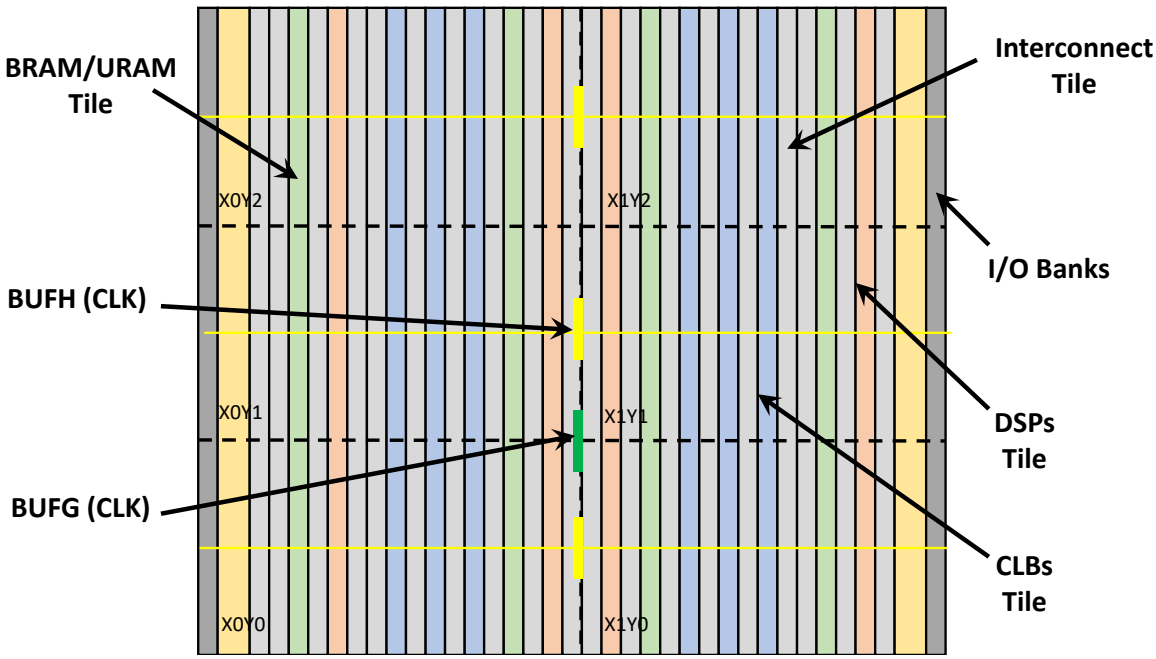


Figure 2.18: A Xilinx Ultrascale FPGA floorplan [131] with several clock regions, each clock region contains a grid of resource tiles for CLB, DSP, BRAMs/URAMs and a grid of interconnects for connection between them.

pre-adder/subtractor, multiplier, and post-adder/subtractor to perform basic arithmetic operations such as addition, and multiplication in a few clock cycles. The FPGA is divided into multiple clock regions, clock buffers are used between them to equally distribute a clock source in the whole FPGA floorplan. Clock buffers are divided into global buffers (BUFG), and horizontal buffers (BUFH). BUFGs are used to forward vertically the clock line to BUFHs which drive clocks to clock regions. Clock regions contain a clock management tile that hosts a phase-locked loop and a mixed-mode clock manager to control the frequency of input clock signals. Moreover, the clock region contains I/O banks to interact with FPGA external peripherals. FPGAs can be programmed by describing the logic of LUTs and configuring the FPGA floorplan interconnect. A logic function can be modeled using hardware description languages (HDLs) that can be synthesized into an RTL netlist. Afterward, a place and route process is required to physically place the netlist to the FPGA resource. Finally, a bitstream generation is conducted to be loaded to the FPGA configuration memory in order to physically configure the FPGA with certain logic functions.

2.3.1 Partial Reconfiguration

Recent families of Xilinx SRAM-based FPGA devices offer the possibility of adapting the device logic at runtime using dynamic partial reconfiguration (DPR) techniques [132]. Partial reconfiguration depicts the configuration of a certain partition of the FPGA floorplan. Each partition hosts a specific configuration of a certain logic function. Therefore, partial reconfiguration allows changing of partition configuration/functionality at runtime without the need to update the whole FPGA configuration [133]. Recently, Xilinx has renamed DPR as dynamic function exchange (DFX) to elaborate the main feature of changing a certain partition functionality

at runtime. The concept of DPR is the exchanging of hardware modules on a certain FPGA partition at runtime. From a system-level point of view, it is similar to the time multiplexing of hardware modules, where each hardware module is active during a specific period based on an application dataflow graph. In order to apply DPR, the FPGA floorplan is split into a static partition and several reconfigurable partitions (RPs) to host multiple reconfigurable modules (RM). RPs can span from a single FPGA frame of resources to multiple clock regions based on the resource requirement of the largest hardware module to be hosted by this partition. A partial bitstream is generated for each RM for the associated RP. Every single RP has a set of partial bitstreams for every hardware module to be hosted by it. Partial bitstreams are loaded to the FPGA configuration memory through dedicated configuration interfaces.

Xilinx FPGAs have several modes of internal and external configurations, based on the selection of the user and system-level requirements. Xilinx internal configuration access port (ICAP) is typically used to control the process of partial bitstream loading internally where the ICAP is located physically on the FPGA fabric. Similarly, for Zynq devices, the processor configuration access port (PCAP) is managing the reconfiguration process internally from the processing system (PS) side. In addition, JTAG is acting as the main external configuration interface to load a full or a partial bitstream to the FPGA configuration memory. Practically, internal configuration interfaces provide a high speed of configuration compared to external ones due to the achievable high data rate.

Many applications already take benefit from partial reconfiguration techniques to apply some sort of runtime adaptability to their functionalities. Software-defined-radio is a well-known use case that takes benefit from DPR to switch between multiple wireless communication standards at runtime reusing the same hardware resources on the FPGA floorplan [134]. [135]. Moreover, signal processing and computer vision-based applications such as radar-based object detection, smart cars, robotics, and wearable devices can also benefit from DPR [136], [137], [138]. Also, for security and cryptography domains, DPR can be used to dynamically swap between several encryption algorithms [139], [140]. Recently, FPGA-based accelerators for machine learning algorithms adopt DPR techniques to support runtime adaptability based on real-time application requirements [141], [142]. In such applications, FPGAs with DPR features can support the execution of multiple application tasks on demand. Instead of physically implementing all required tasks by an application on a single large FPGA, smaller-size FPGAs can be used with less resource and power consumption to implement tasks temporarily at runtime. Heterogeneous MPSoC can also benefit from DPR by exchanging and upgrading the processing elements or custom hardware accelerators at runtime based on workload requirements. In this context, several research works have proposed multiple frameworks and platforms for runtime reconfigurable MPSoC.

A reconfigurable MPSoC platform based on Xilinx Zynq devices is proposed by [143] for space applications. This reconfigurable platform is based on static on-board processors with reconfigurable loosely coupled multi-accelerator architecture. The platform provides runtime adaptability to contribute to the full system fault tolerance to support the main requirements for space applications. The multi-accelerator architecture is based on the ARTICO3 framework [144]. The platform supports a scalable number of hardware accelerators, where each accelerator is hosted by an accelerator tile with local memory and register file for controlling. Each accelerator tile hosts a reconfigurable partition for the accelerator logic to be modified at runtime using DPR. Moreover, the platform supports a real-time operating system running on the processing side of the Zynq device to manage and control the system including the reconfiguration process of the multi-accelerator architecture.

On the other hand, DPR can be used in high levels of granularity such as FPGA overlays as proposed by [145]. This work exploited the DPR technique to build a dynamically multi-grain reconfigurable and scalable overlay architecture. The proposed overlay consists of multiple small-size RPs that can be reconfigured at runtime to map several applications with different requirements. The overlay size is changeable and it can be integrated with a host processor or with other hardware accelerators. An automated toolflow is developed to automatically offload kernels to the reconfigurable overlay. Despite, the great importance of DPR to develop and implement adaptive FPGA-based SoC or reconfigurable multi-core architectures on FPGAs. The main challenge remains the reconfiguration management, and how to reduce the reconfiguration time to meet real-time application requirements. Also, the abstraction of the reconfiguration process from the software layer requires an efficient and reliable reconfiguration management method [146].

2.3.2 Reconfiguration Management Frameworks

The DPR management from the software side imposes several challenges related to the abstraction of the reconfiguration process from the CPU side, in addition to the reconfiguration efficiency in terms of reconfiguration time especially in cases of real-time applications [147]. Consequently, DPR management requires a custom hardware implementation for a DPR controller between the CPU and the dedicated programmable region. Besides, the management of the hardware-accelerated modules on the PL from the software running on the CPU. Therefore, a change in adaptive systems requirements (e.g. hard real-time scenarios), or adoption of a new instruction set (e.g. RISC-V) requires the development of new software drivers or overlays along with updates to the hardware implementation of the DPR controller for compatibility purposes.

In the last years, several works are proposed by literature for DPR controllers including custom DPR controllers or software abstraction layers for DPR management. In [148], the authors proposed a high-speed DPR controller for Xilinx FPGA devices. The controller's system architecture is designed for loading partial bitstreams from an off-chip memory to the FPGA configuration memory at a data rate close to the physical data rate of Xilinx's ICAP primitive. The proposed controller uses DMA components for data transfer, freeing the adaptive SoC's CPU to execute other tasks. Similarly, the ZyCAP manager [149] features a high throughput DPR for Xilinx Zynq FPGAs, customized for ARM processors hosted on the PS side of the Zynq device. ZyCAP provides a set of high-level driver interfaces to manage the reconfiguration process from the PS side. However, ZyCAP is exclusively compatible with Xilinx Zynq FPGAs and its portability to other devices requires hardware and software modifications. Meanwhile, the proposed DPR manager by Carlo et al. [150] provides portability to several Xilinx FPGA devices with an ICAP interface.

Furthermore, it supports a safe DPR for real-time and mission-critical adaptive applications. As a consequence, a new set of features are required for the implementation of the DPR controller with customization on the operation modes. Therefore, the controller features different modes of operations depending on the application requirements and invokes a cyclic redundancy check and error correction on the loaded partial bitstream before transferring it to the configuration memory. Besides, the DPR process is software managed by a LEON3 soft-core processor.

Table 2.4: State-of-the-art DPR management units comparison.

Ref.	SoC Processor	Support Abstraction Layer	Throughput (MB/s)	Freq. (MHz)
Vipin et al.[148]	Microblaze	✗	399.8	100
ZyCAP [149]	ARM	✓	382	100
Anderson et al. [150]	LEON3	✓	395.4	100
RT-ICAP [151]	Patmos	✓	382.2	100
AC ICAP [152]	Microblaze	✗	380.47	100
Xilinx PCAP [153]	ARM	✗	128	100
RV-CAP [21]	RV64GC (Ariane)	✓	398.1	100

In the same contest, for hard real-time adaptive applications, the RT-ICAP controller [151] is introduced as a time-predictable DPR controller. It aims to reduce the worst-case execution time (WCET) to perform the configuration in a determined amount of time. The controller features the capability of partial bitstream compression before transferring it to the FPGA configuration memory to reduce its size and therefore reduce the reconfiguration time. However, extra on-chip memory is reserved on the FPGA fabric to store the compressed partial bitstream. The DPR process is software managed through a custom real-time soft-core processor. It is used for encryption and security applications. Therefore, AC ICAP [152] is introduced as a light DPR controller that can operate autonomously or with a light soft-core processor (e.g. Microblaze). The controller is customized for partial reconfiguration of LUT resources only featuring low resource overhead and high reconfiguration throughput. However, it lacks the portability for the new generation of FPGA architectures.

Accordingly, from the abovementioned work, designing DPR controllers depends on one side on the timing characteristics and reconfiguration sensitivity of the target applications as well as the architecture of the hardware/software co-design of the target adaptive platform. Several DPR controllers are proposed to support the management of the reconfiguration process from operating systems running on application class processors (i.e., ARM processors) to enhance software productivity. Hence, suitable software drivers and interfaces between the CPU and the FPGA are required. Al Kadi et al. [154] proposed a set of software drivers running on Linux for Xilinx Zynq devices to access the processor configuration access port (PCAP). Another novel approach is called Pynqpartial [155]. This is introduced as a software-only implementation for managing DPR from the Pynq platform. Thus, a set of Python packages are implemented on the ARM processor on the PS side using the existing PCAP interface to access the configuration memory. However, the pynqpartial shows a poor reconfiguration throughput. Meanwhile, the authors of [156] improved the reconfiguration throughput while maintaining a high level of abstraction for DPR management from a Petalinux operating system targeting a Xilinx Zynq Ultrascale+ FPGA.

Table 2.4 shows a comparison between pre-described state-of-the-art DPR management units including the proposed reconfiguration management in this PhD thesis (RV-CAP) [21]. The RV-CAP unit is mainly developed to support DPR management for FPGA-based RISC-V SoC. It consists of a DPR controller directly connected to a RISC-V processor through a

shared bus interconnect, and a set of software drivers to abstract the reconfiguration process through software functions running on a RISC-V processor. The proposed reconfigurable management is integrated into the main processing tile of the proposed heterogeneous many-core architecture to support the runtime adaptation feature to change types and configurations of many-core tiles at runtime. RV-CAP is completely implemented using custom hardware modules controlling the ICAP primitive with the ability to be portable for any Xilinx FPGA devices that support DPR. It features a small area footprint with a small number of resource utilization on the FPGA which makes it suitable for small sizes FPGAs as well.

2.4 Contribution Towards Modular and Adaptive Many-Core Architectures

In this section, the state-of-the-art tile-based many-core architectures and hardware accelerator integration presented previously are compared to the main contributions of this dissertation. As mentioned before in the introduction section, this PhD thesis has three main contributions:

- Modular many-core architecture to support heterogeneous ISAs for general purposes workloads.
- Seamless integration of custom hardware accelerators through a hybrid tile architecture for accelerators and memory modules.
- An internal reconfiguration management unit to support self-adaptation at run-time for several heterogeneous many-core configurations.

The first and third contributions together represent the proposed modular and adaptive heterogeneous tile-based many-core architecture [16]. The proposed architecture is based on a NoC-based architecture by utilizing the ARTNoC NoC framework proposed by [11]. The proposed tile-based architecture is based on a modular and parametrized implementation that supports single and multi-core general-purpose architecture. The tile also features a modular memory hierarchy that can be tailored to support a non-coherent shared memory multi-core architecture or a hybrid memory hierarchy with additional scratchpad memory per core [17]. The second contribution focuses on the seamless integration of custom hardware accelerators into the proposed tile-based many-core architecture as a LCA tile attached to the NoC [18]. Moreover, the LCA tile supports the feature of accelerator memory reuse. Therefore, general-purpose tiles are able to access and reuse the on-chip PLM of the accelerator tile as a scratchpad shared memory unit between them through the NoC.

2.4.1 Modular and Adaptive Heterogeneous Tile-based Architecture

Several research approaches have been proposed for adaptive and self-aware many-core systems to allow the re-usability and reconfigurability of many-core architectures to be adjusted according to multiple requirements for different application domains. As a result, an expected reduction in development time and cost can be achieved by the adoption of adaptive many-core approaches. However, adaptive many-core approaches require a sort of

modularity of hardware components to ensure proper integration and communication between them after the adaptation process. In tile-based many-core architectures, modularity can be achieved first by using a unified communication method between heterogeneous compute tiles through a NoC, or advanced bus-based architectures. In addition, a heterogeneous set of compute tiles that share the same inter-tile interfaces and apply the same communication protocol over the many-core communication medium as well as a unified parallel programming model are supported. Therefore, this dissertation proposes an adaptive and heterogeneous tile-based architecture to accommodate modularity and adaptability to reduce design and integration time and promote the commodity of many-core architectures for emerging application domains.

Table 2.5 shows a comparison between the aforementioned tile-based many-core architectures presented in (Section 2.1) and the proposed adaptive and heterogeneous tile-based architecture (AGILER). Accordingly, state-of-the-art comparison in Table 2.5 is based on the following points:

- Level of heterogeneity by supporting multiple ISAs and custom hardware accelerators.
- Modularity and architecture characteristics by supporting different microarchitecture configurations and memory hierarchy.
- Architecture configurability and prototyping

Heterogeneity level: Several works propose multi ISA for general-purpose computing tiles. ESP [13], and Manticore [62] support two RISC-V ISAs in a similar manner to the proposed many-core architecture in this dissertation (AGILER). The two RISC-V ISAs are based on RV32 and RV64 ISAs, where compute tiles can be configured to support one of them based on target workloads requirements. The rest of the work and platforms presented in Table 2.5 support only one ISA for their general-purpose tiles as shown in the second column.

Moreover, custom hardware accelerators are mostly supported by all heterogeneous many-core architectures, the main difference lies in the accelerator coupling model and the level of granularity. In other words, where hardware accelerators are coupled and reside inside the many-core architecture, either as a separate LCA tile or as a shared accelerator peripheral within a general-purpose tile. TaPaSCoc [55], Savas et al. [159], BlackParrot [50] , ESP [13], and Memphis [47] support multiple LCA tiles where a custom hardware accelerator can be hosted by a separate tile and communicate to the rest of the system via a scalable and high bandwidth communication fabric (e.g. NoC, cascaded bus interconnect). On the other hand, Hero [157], GRVIPhalanx [38], P2012 [51], and RVNoC [162] support intra-tile shared hardware accelerators that are integrated within general purpose tiles as shared memory-mapped peripherals. In this type, shared hardware accelerators can be accessed by any of processing cores inside the tile through a shared interconnect (e.g. bus-based interconnect, point-to-point communication). Accordingly, the proposed tile-based many-core architecture by this dissertation supports both ways of custom hardware integration either inside general purpose tile as shared memory-mapped peripherals or hosted by a separate tile as a LCA.

Modularity and microarchitecture: Design modularity and microarchitecture define the internal structure of any tile-based many-core system and the degree of flexibility and reusability to be tailored to implement several many-core taxonomies based on target application requirements. In order to evaluate that, an analysis of tile microarchitecture has been conducted as shown in Table 2.5 (columns 4, 5, and 6) regarding supported memory hierarchies, scalability, number of cores, and interconnection type. Therefore, several works feature a

Table 2.5: Many-core architectures State-of-the-Art comparison.

Many-Core Architecture [Ref.], (Year)	Heterogeneity Level		Architecture Charecteristics		Memory Hierarchy per Tile	Configurability	Proto-type
	Support Custom HW Integration	Support Multiple ISAs	Tile-based Architecture	Communication/ Interconnection			
Hero [157] (2017)	✓	✗	Multi-Core	Bus-based	Shared Memory	Design-Time	FPGA
OpenPiton+Ariane [43] (2019)	✗	✗	Single-Core	NoC	Local Memory	Design-Time	FPGA
Andromeda [158] (2021)	✗	✗	Multi-Core	Bus-based+NoC	Shared Memory	Design-Time	FPGA
MemPool [56] (2021)	✗	✗	Multi-Core	Cascaded Crossbar	Shared Memory	Design-Time	ASIC
TaPaSCo [55] (2019)	✓	✗	Multi-Core	Cascaded Crossbar	Local Memory	Design-Time	FPGA
Savas et al. [159] (2020)	✓	✗	Single-Core	NoC	Local Memory	Design-Time	FPGA
GRVIPhalanx [38] (2016)	✓	✗	Multi-Core	Crossbar+NoC	Local/Shared Memory	Design-Time	FPGA
BlackParrot [50] (2020)	✓	✗	Single-Core	NoC	Local Memory	Design-Time	ASIC
Manticore [62] (2021)	✗	✓	Multi-Core	Cascaded Crossbar	Shared Memory	Design-Time	ASIC
ESP [13] (2020)	✓	✓	Single-Core	NoC	Local Memory	Design-Time	FPGA
P2012 [51] (2012)	✓	✗	Multi-Core	NoC	Local/Shared Memory	Design-Time	ASIC
CoreVA-MPSoC [160] (2018)	✗	✗	Multi-Core	NoC	Local/Shared Memory	Design-Time	ASIC
Vestias et al. [161] (2015)	✗	✗	Single-Core	Bus-based	Shared Memory	Design-Time	FPGA
Memphis [47] (209)	✓	✗	Single-Core	NoC	Local Memory	Design-Time	FPGA
RVNoC [162] (2018)	✓	✗	Single-Core	NoC	Local Memory	Design-Time	FPGA
Epiphany [163] (2014)	✗	✗	Single-ore	NoC	Local Memory	Design-Time	ASIC
Kalray MPPA256 [164] (2013)	✗	✗	Multi-Core	NoC	Local Memory	Design-Time	ASIC
This dissertation (AGILER) [16] (2022)	✓	✓	Multi-Core	NoC	Local/Shared Memory	Runtime	FPGA

high degree of scalability to support scalable numbers of multiple types of heterogeneous compute units such as Hero [157], OpenPiton [43], MemPool [56], TaPaSCo [55], BlackParrot [50], GRVIPhalanx [38], Manticore [62], and Kalray [164]. They are able to support hundreds of processing cores clustered in tens of tile-based architectures. Also, several works support different memory hierarchies within a single-tile architecture. They feature a shared memory hierarchy between tile's PEs, where each PE has its own local memory either a scratchpad memory or a cache memory such as GRVIPhalanx [38], P2012 [51], and CoreVA-MPSoC [160]. The proposed adaptive and heterogeneous tile-based architecture satisfies the compute performance scalability using a scalable mesh-based NoC topology for inter-tile communication with a variant set of heterogeneous compute tiles. Each compute tile features a configurable multi-/single-core architecture that can be configured with variant numbers and types of RISC-V-based PEs. Moreover, shared and local memory hierarchies with parameterized sizes are supported per each compute tile. Therefore, the proposed architecture provides the flexibility for tailoring several many-core configurations for compute or memory-bound applications.

Configurability and prototyping: The proposed many-core architecture supports run-time adaptation through an internal reconfiguration manager (RV-CAP) using dynamic and partial reconfiguration technology on Xilinx FPGAs. The modularity and adaptability features of the proposed architecture allow the flexibility to be ported to other Xilinx FPGA series. Accordingly and to the best of our knowledge, the proposed architecture is the first heterogeneous tile-based many-core architecture for multiple ISAs and custom hardware accelerators that supports self-adaptation using an internal run-time DPR manager for several heterogeneous many-core configurations on FPGAs. According to state-of-the-art comparison presented in Table 2.5 (column seven), all tile-based many-core architectures are design time configurable with a limited degree of portability to different hardware platforms in case of ASIC implementation. Few of them have the capability to be portable to different FPGA devices such as HERO [157], ESP [13], TaPaSCo [55], and OpenPiton [43].

2.4.2 Hybrid Memory/accelerator Tile Architecture

The LCA model is increasingly used in heterogeneous architecture to achieve an order of magnitude high computing performance. However, LCAs require a large portion of private local scratchpad memory with the accelerator logic inside the custom accelerator tile architecture. Accordingly, the increasing number of accelerator tiles leads to a significant increase in accelerators' PLM resources. For FPGA-based manycore systems, block memories BRAMs are used to implement PLMs which have limited availability on FPGAs. Therefore, memory sharing between accelerator tiles and general-purpose tiles is necessary to reduce many-core systems' memory footprint. Recent heterogeneous SoCs are characterized by a large number of hardware accelerators coupled with many general-purpose compute units in a so-called accelerator-rich architecture. These types of heterogeneous SoCs provide the capability to adapt their architectures to specific application workloads aiming to increase performance and energy efficiency.

Accordingly, several research approaches have been proposed to seamlessly integrate accelerator and memory tiles into manycore systems to reduce design costs and increase architecture reusability. Table 2.6 shows a comparison between aforementioned state-of-the-art

Table 2.6: Accelerator Integration State-of-the-Art Comparison.

Ref. (Year)	HW Acc. Coupling Model	Support Acc. Memory Reuse	Support Coherency	Support Large PLM	Prototype
Ng et al. [105] (2016)	Tightly Coupled	✗	✓	✗	Simulation
Bellochi et al. [121] (2021)	Loosely Coupled	✗	✓	✗	FPGA
Savas et al. [159] (2020)	Loosely Coupled	✗	✗	✓	FPGA
Mantovani et al. [119] (2016)	Loosely Coupled	✗	✓	✓	FPGA
Dehyadegari et al. [120] (2015)	Loosely Coupled	✗	✓	✓	Simulation
Li et al. [122] (2011)	Loosely Coupled	✗	✗	✓	Simulation
Pilato et al. [127] (2017)	Loosely Coupled	✗	✓	✓	Simulation
Cota et al. [129] (2016)	Loosely Coupled	✓	✓	✓	Simulation
Cong et al. [128] (2012)	Loosely Coupled	✓	✓	✓	Simulation
Kamaleldin et al. [18] (2022)	Loosely Coupled	✓	✗	✓	FPGA

accelerators integration models in (Section 2.2) and the proposed hybrid memory/accelerator tile architecture. Accordingly, state-of-the-art comparison in Table 2.6 is based on the following points:

- Hardware accelerator coupling model
- Supporting accelerator memory reuse and sharing accelerator PLM with other computing tiles in the target system
- Supporting coherency and large accelerator PLMs

Accelerator coupling model: Several works propose different ways of accelerator coupling with general-purpose compute units. As discussed in (Section 2.2.1), there are two ways for accelerator coupling either as a TCA or LCA to general-purpose processors. Therefore, Table 2.6 (column two) shows hardware accelerator coupling models of several related works. Most of the related work supports LCA model except Ng et al. [105] which supports ISA extension for tightly coupled accelerators to processor pipelines. State-of-the-art LCA models feature a

separate tile to host hardware accelerators either as HLS-based accelerators such as Pilato et al. [127], Mantovani et al. [119], and Cota et al. [129]. On the other hand, Savas et al. [159] supports Chisel-based accelerators within the Rocketchip generator framework. In addition, LCAs within a general-purpose tile are supported by Dehyadegari et al. [120], and Bellochi et al. [121].

In this case, LCAs are shared memory-mapped peripherals to general-purpose cores within compute tiles. Accordingly, the proposed hybrid memory/accelerator tile architecture [18] supports a LCA integration model where hardware accelerators can be integrated inside a LCA tile. The proposed LCA tile is connected to the NoC through a dedicated NI that allows data transfer between the tile's PLM and other compute tiles within the proposed many-core architecture. The proposed LCA tile supports the integration of both RTL-/HLS-based accelerators with various on-chip memory sizes for PLM realization.

Accelerator PLM sharing: The second feature of the proposed hybrid memory/accelerator tile is accelerator memory reuse. As described in (Section 2.2.2), accelerator memory reuse is an important feature proposed by several related works. The main idea is to extend the many-core shared memory portion by partially or fully reusing accelerators PLMs during their inactive time. In this context, Table 2.6 (column three) shows related work LCAs that support the feature of accelerator memory reuse. Cota et al. [129], and Cong et al. [128] firstly introduced the concept of accelerator memory reuse as an L2 shared cache memory to general-purpose tiles. They only use part of the PLM with extra control logic for cache management implemented on each LCA tile. Despite the efficient utilization of LCAs PLMs, an extra implementation and design effort is required to modify the LCA tile to operate as an extended cache memory or a pure hardware accelerator. Therefore, the proposed hybrid memory/accelerator tile provides more design flexibility with less development effort to support memory and acceleration mode of operation through a parametrized architecture that can be configured through a set of configuration messages from any general-purpose tiles within the proposed architecture.

2.5 Summary

In this chapter, background and state-of-the-art tile-based many-core architectures, hardware accelerator integration, and adaptive FPGA-based SoC are presented, analyzed, and evaluated with respect to the main contributions of this dissertation.

Section 2.1 describes in detail homogeneous and heterogeneous tile-based many-core architectures and their characteristics. Recently, tile-based many-core architectures are being at the centre of the compute-centric paradigm. They represent the evolution from single-core architectures to hundreds of cores on the same chip or within a single SoC. Tile-based many-core architectures consist of a large number of cores with several levels of memory hierarchies and interconnect. Moreover, Section 2.1.1 shows various state-of-the-art tile-based platform implementations. Those platforms are characterized by many architectural features such as supporting multi ISAs, supporting heterogeneous hardware accelerators, different memory hierarchies, and high degrees of design modularity and extensibility. Afterward, Section 2.1.2 shows a classification of open-source RISC-V ISA.

Three classes of RISC-V-based processors are presented spanning from little cores to high-performance cores. In this dissertation, RISC-V-based processors are used to implement general-purpose cores.

Section 2.2 explains hardware accelerator integration in terms of accelerator coupling models and accelerator memory interactions within many-core and SoC architectures. LCA and TCA integration models are presented in Section 2.2.1 and their related state-of-the-art is described and analyzed in detail. In addition, accelerator memory interaction is presented in Section 2.2.2. Several accelerator memory interaction techniques and supported features are presented and analyzed based on direct memory access and coherency, supporting large data sets, and accelerator memory reuse.

Runtime adaptive FPGA-based SoC is described in Section 2.3 including an overview of modern FPGAs structure. Section 2.3.1 presents and analyses partial reconfiguration techniques for Xilinx FPGAs. Also, the utilization of dynamic partial reconfiguration techniques in recent adaptive MPSoC architectures and configurable FPGA overlays. Afterward, reconfiguration management units are presented in Section 2.3.2. A comparison between the state-of-the-art DPR management units and the proposed RV-CAP unit is presented.

Finally, Section 2.4 presents the main dissertation contributions toward modular and adaptive many-core architectures and state-of-the-art comparison with the dissertation's proposed solutions and approaches.

3 A Modular Tile-based Many-Core Architecture for Heterogeneous ISAs

Machine learning and data-centric applications constitute the main driving forces for computing's rapid evolution. Over the past decade, several computing paradigms have been introduced seeking to increase computing performance scaling and energy efficiency in order to cope with the emergence of new application classes with massive and irregular data sets. Among those computing paradigms are compute-centric architectures which are still leveraged in the mainstream multi-/many-cores SoC developed by industry and academia for several application domains. Compute-centric systems went through a tremendous evolution from multi-core homogeneous architectures to highly heterogeneous architectures with big, and little cores as well as application-specific accelerators.

Despite the high-performance gain of heterogeneous architectures, the increasing numbers of heterogeneous elements are limited by the system interconnects scalability and therefore the degree of compute performance scalability. This obstacle of compute-performance scaling is referred to as the scalability wall. Therefore, tile-based architectures are developed for highly scalable many-core systems with a growing capacity of heterogeneous compute elements. The degree of scalability for tile-based architectures relies on the inter-tile communication fabric which on recent many-core approaches depends on scalable NoC variant topologies. However, the design and development of tile-based many-core architectures is a cumbersome process in terms of development time and costs. Especially if target application domains require a high capacity of heterogeneous compute tiles as is the case in recent computing devices that support a wide range of application domains.

As a result, integrating more components and different architectural units on a complete system-on-chip increases design efforts (e.g. verification, validation, integration) and therefore the development time and costs. Moreover, the design specifications could vary due to different application requirements which lead to the necessity of a new design process for each new application requirement. Resultantly increasing the design effort and therefore time to market with continuous inflation in non-recurring engineering costs.

This chapter presents a modular tile-based many-core architecture to support heterogeneous ISAs through the adoption of multiple general-purpose RISC-V based cores with different ISAs. In this context, this chapter introduces a novel modular implementation for many-core architectures based on RISC-V open-source-hardware processors with tile-granularity customization for FPGA platforms. The proposed tile-based many-core design has re-usable

and flexible architectural units that can be tailored to implement different heterogeneous and homogeneous many-core taxonomies using regular building blocks for computation (i.e. PEs, compute tiles), with several memory hierarchies and generic communication interconnections. The tile-based architecture maintains a high degree of scalability using a scalable NoC topology and the design regularity manner offers the flexibility to scale up the number of compute tiles with less design effort and cost. Furthermore, a message-based communication model is adopted to support data transfer over the NoC between compute tiles. In addition, a bare metal programming method is introduced on the level of compute tiles for parallel programming over the RISC-V-based PEs using shared and local scratchpad data memories on the compute tile level. Moreover, the proposed tile-based many-core architecture is evaluated based on different architecture configurations covering different types of memory hierarchies/sizes, communication interconnections, and numbers/types of tiles/cores per many-core system to explore several design choices and their effects on the system performance. The tile-based architecture is implemented and evaluated on a Xilinx Virtex Ultrascale+ FPGA.

The Chapter is structured as follows. Section 3.1 presents the hardware architecture of the tile-based many-core architecture including several types of general-purpose single-/multi-core compute tiles, different RISC-V-based PEs supporting different RISC-V ISAs with tightly-coupled local scratchpad memory. Also, tile modularity and configurability during design time to support different memory hierarchies and several types of PEs are described. Section 3.2 presents how to achieve a highly scalable tile-based architecture using a parametrized NoC architecture with a customized communication model protocol for data transfer between heterogeneous compute tiles. Section 3.3 presents the supported bare-metal programming method for parallel task execution over the tile-based architecture. In addition, memory partitioning of a single compute tile is described in order to specify the used type of memory (i.e. shared or local) and the target PE for parallel execution of multiple kernels. Section 3.4 presents the evaluation and prototyping of the tile-based many-core architecture using several tile-based configurations in terms of hardware resource utilization, computing performance scalability, achievable memory bandwidth, and communication data rate between compute tiles with several signal processing and neural network benchmarks. Finally, Section 3.5 summarizes this chapter.

3.1 Modular Tile-based Architecture

The tile-based many-core architecture features a modular and hierarchical interconnect design that targets domain-specific and general-purpose applications for FPGA accelerators. Moreover, the tile-based many-core architecture can be considered as a model for rapid prototyping of different many-core taxonomies with homogeneous or heterogeneous computing elements (multi ISAs and application-specific hardware accelerator cores) and supports different styles of interconnect topologies. The tile-based architecture consists of a scalable number of compute tiles connected by a Network-on-Chip interconnect as shown in Figure 3.1. The compute tile architecture can be configured to support two RISC-V ISAs (RV32, RV64). Moreover, compute tiles can be configured to support single-core and multi-core architectures with the flexibility to support several memory hierarchies. Each compute tile features a private address space which allows communication between all PEs and shared

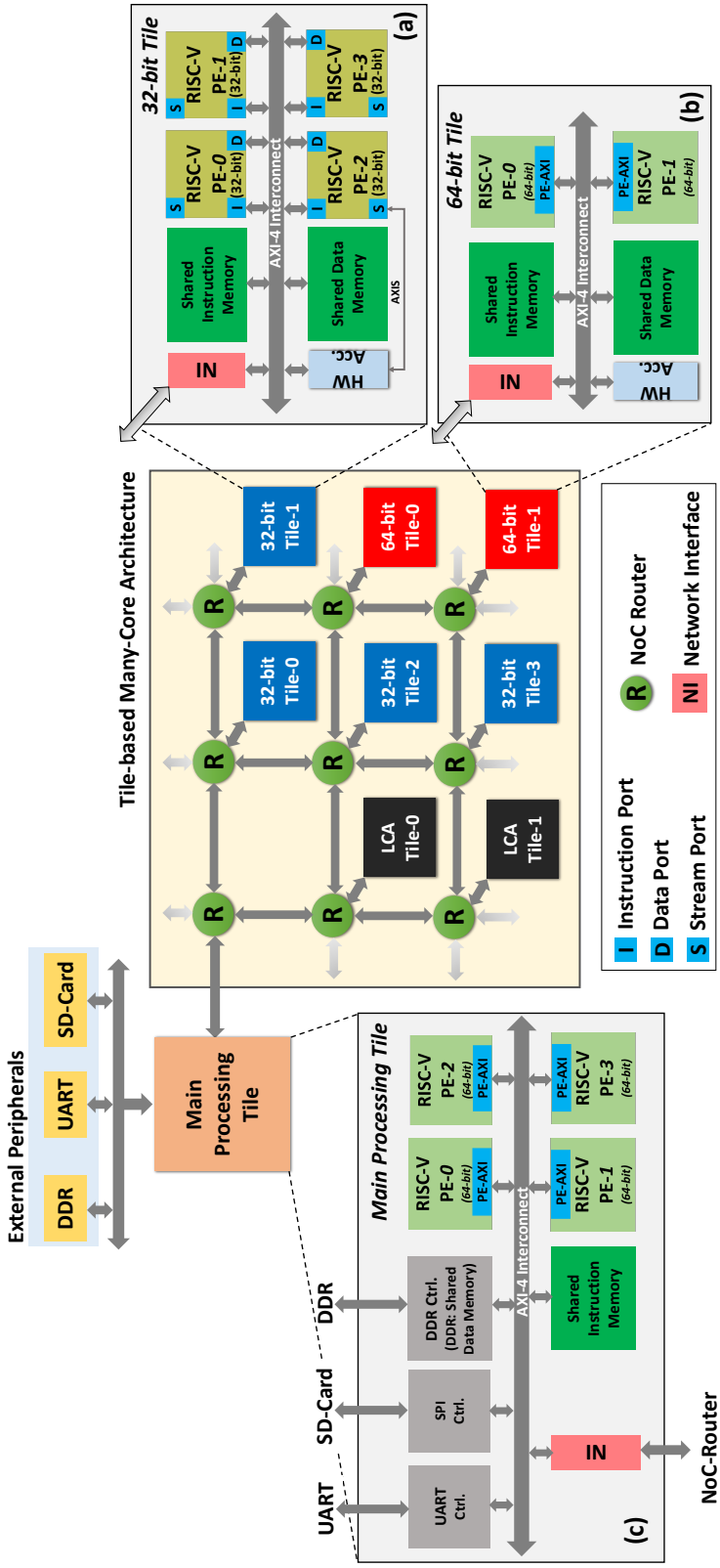


Figure 3.1: Overview of the modular tile-based many-core architecture with a 3x3 tile-based many-core configuration including (a) 4x32-bit general-purpose compute tiles, (b) 2x64-bit general-purpose compute tiles, (c) the main/primary processing tile, and heterogeneous tiles to host custom hardware accelerators (LCA tiles).

tile peripherals through shared data memory via an AXI interconnect. In this section, the tile-based many-core architecture and its architectural components are described.

3.1.1 Multi-Core based Tile Architecture

General-purpose compute tiles are the core of tile-based architecture, they represent the computing nodes for the proposed many-core system. As shown in Figure 3.1, the tile-based many-core architecture consists of three types of heterogeneous tiles that support multiple RISC-V ISAs in addition to custom hardware accelerators (LCA tiles). All general-purpose compute tiles have a regular design pattern based on a bus-based architecture. The compute tile tightly couples single or multiple RISC-V based PEs with shared instruction and data scratchpad memories using a shared AXI interconnect. Therefore, all PEs in a single tile share a common private address space. The shared bus architecture allows the communication between RISC-V-based PEs and tile's shared memories as well as memory-mapped peripherals via AXI interconnect.

To enhance the memory bandwidth, shared instruction and data memories are implemented using dual-ported BRAM/URAM blocks. Therefore, two memory read/write (R/W) channels can be established across AXI interconnect to handle two memory requests simultaneously. Shared instruction memory is implemented as read-only BRAM memory which is used as a boot memory during the memory initialization stage to load the compiled binary file for execution. Each compute tile implements a uniform memory access (UMA) architecture, where each RISC-V-based PE can access shared data and instruction scratchpad memories connected to the AXI interconnect as a slave memory-mapped peripheral. In the UMA architecture, each PE experiences the same bandwidth and access latency to the memory. However, the overall memory bandwidth is divided between the number of PEs per tile. Therefore, the growing number of PEs connected to the AXI interconnect leads to an increase in memory access latency per each PE and increases the probability of memory congestion due to limited AXI interconnect bandwidth.

In order to reduce memory congestion per tile, we use an open-source high-performance coherent AXI interconnect implementation [63], [165]. The AXI interconnect is based on a fully-connected crossbar where each slave port has a dedicated connection to each master port. The crossbar supports up to five independent data transaction channels for R/W and applies a round-robin arbitration scheme. However, the memory bandwidth scalability is limited and starts to saturate after a certain number of PEs. Therefore, each tile supports a maximum number of four PEs to ensure a congestion-free tile implementation.

The three types of general-purpose compute tiles are described as follows:

- 32-bit compute tile: As shown in Figure 3.1 (a), the first compute tile is a 32-bit general-purpose multi-core tile with four RV32 PEs. Each PE consists of a single RV32IMC core with tightly-coupled scratchpad local memory for data and instruction. The RV32 PE is compatible with a 32-bit AXI interface to seamlessly connect it to the tile AXI interconnect. The 32-bit tile hosts shared scratchpad instruction and data memory for application booting and data sharing between PEs. In addition, optional memory-mapped peripherals such as custom hardware accelerators are supported by the tile. The tile is connected to the NoC router through a memory-mapped NI for data transmission and receiving with other many-core tiles.

- **64-bit compute tile**: The second general-purpose compute tile is a single- or dual-core 64-bit tile that can be configured to support single or dual RV64 based PEs as shown in Figure 3.1 (b). Similar to the RV32 PE, the RV64 PE consists of a single RV64IMAC core with tightly-coupled local scratchpad memory for data and instruction. The RV64 PE can be seamlessly connected to the tile interconnect via 64-bit AXI interfaces. The 64-bit tile hosts shared scratchpad instruction and data memory for application booting and data sharing between PEs. In addition, optional memory-mapped peripherals such as custom hardware accelerators are supported by the tile. The tile is connected to the NoC router through a memory-mapped NI for data transmission and receiving with other many-core tiles.
- **Main processing tile**: The third type of general purpose compute tile is shown in Figure 3.1 (c). It is the main and permanent processing tile of the tile-based many-core architecture. The main processing tile is based on a 64-bit quad-core architecture with shared instruction and data memory. The off-chip DDR memory is used as shared data memory for a large capacity of data sharing between the four RV64 PEs, while the shared instruction memory uses on-chip BRAM blocks similar to other compute tiles. Moreover, the main processing tile controls and manages external many-core peripherals (i.e. SD-card, UART) and it can be extended to support other types of off-chip peripherals. Furthermore, the reconfiguration manager unit and its associated components (i.e. direct memory access (DMA), internal configuration access port (ICAP) controller) are hosted and managed inside the main processing tile as described in Chapter 5. Therefore, the main process tile is the permanent general-purpose tile for the proposed tile-based many-core architecture which is primarily responsible for many-core management and configuration as well as taking part of computational workload with other tiles. The main processing tile is equipped with a generic NI to the NoC router for inter-tile communication.

3.1.2 Heterogeneous RISC-V based Processing Elements

The Processing Element is the main computing unit inside the proposed tile-based many-core architecture. The inherent design modularity of the PE allows the execution of general-purpose applications across different domains e.g. (signal or image processing, and machine learning) with different computing requirements and memory footprints. The PE consists of a single open-source RISC-V soft-core processor and a local tightly coupled memory (TCM) subsystem for data and instructions to increase data locality for compute- and memory-intensive applications. In addition, like typical Harvard architecture, the PE features separated local instruction and data memories tightly coupled with the RISC-V soft-core processor. The local TCMs feature a low memory latency of one clock cycle for R/W operations for private computation within a single PE. Moreover, using a local memory per each PE reduces the probability of memory interference between multiple PEs compared to the UMA in shared memory hierarchies. In this section, two PEs based on RV32/RV64 ISAs for 32-/64-bit compute tiles are described as follows.

32-bit PE

The 32-bit PE consists of a single open-source RI5CY soft-core processor [77] with an implemented local tightly coupled memory subsystem for data and instructions as shown in Figure 3.2. RI5CY core is a 32-bit 4-stage pipeline in-order processor. The core implements a simple

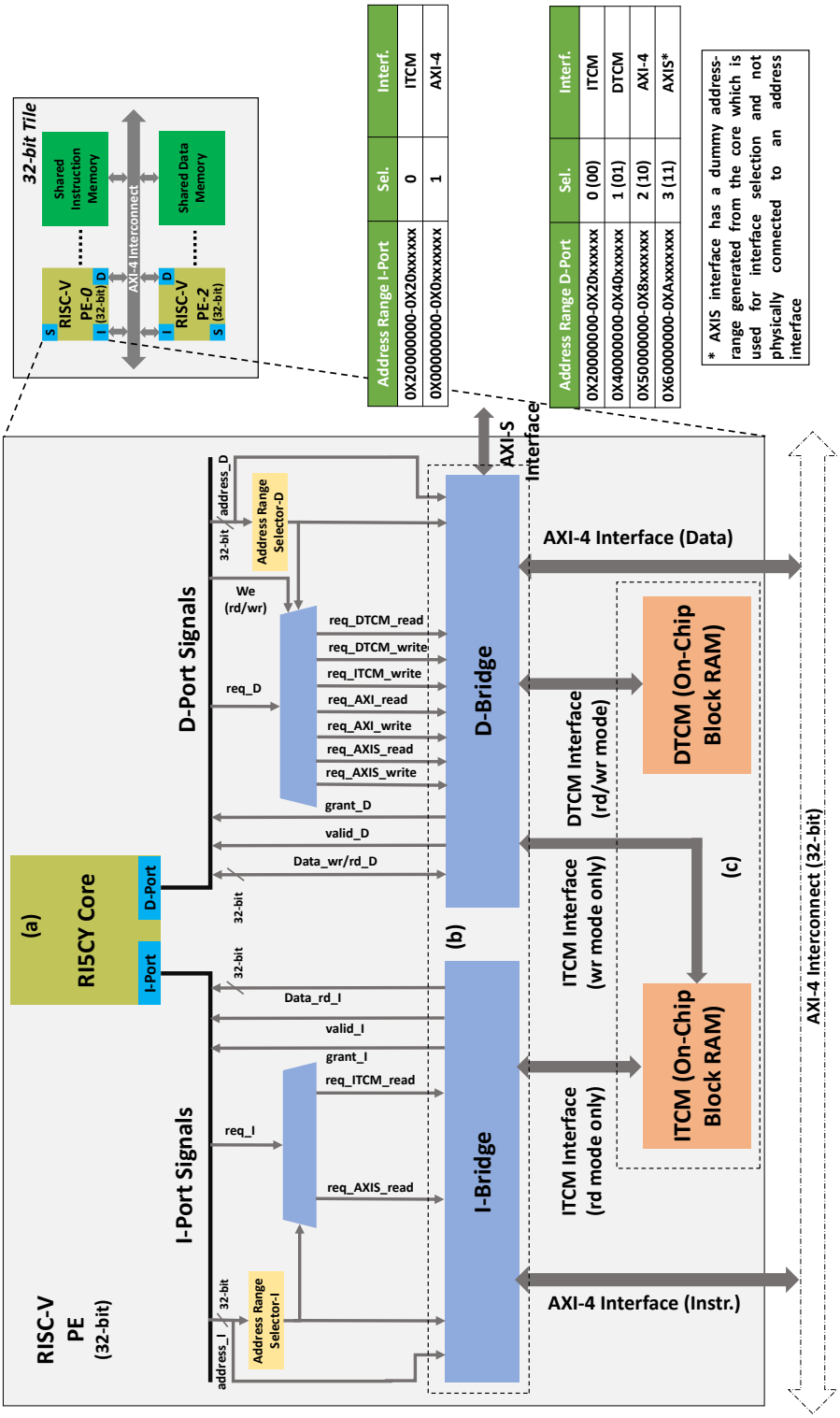


Figure 3.2: Schematic of 32-bit RISC-V based PE showing: (a) open-source RV32IMC (RI5CY) core, (b) instruction and data bridges for converting native I/D signals to AXI-4 interfaces, (c) on-chip I/D TCM and their connection to the RI5CY core through I/D bridges.

RV32IMC ISA with a main arithmetic-logic unit (ALU) and dedicated units for multiplication, division and multiply-accumulate (MAC). The average base instructions loading latency from instruction memory is one clock cycle except for load/store (LD/ST) instructions and other custom instructions which have a minimum latency of 2 clock cycles [166].

The PE features 2 separate tightly coupled memory blocks implemented using on-chip BRAM/URAM for instruction and data (I/D) as shown in Figure 3.2 (c). I/D-TCM offer low memory latency of one clock cycle for R/W operations, it also increases data locality for memory-bound applications. All memory blocks have a fixed word size of 32-bit compatible with RV32 ISA. Besides the RI5CY core, the PE local memory subsystem and external interfaces are described as follows.

- 32-bit ITCM: As shown in Figure 3.2 (c), the ITCM is implemented as a dual-ported on-chip BRAM/URAM with a read-only interface to the RI5CY core instruction port (I-Port) for fetching a new instruction every single clock cycle. In addition, a write-only interface to the data port (D-Port) allows the transfer of specific instructions from the shared instruction memory to the ITCM during the memory initialization stage. In this case, the RI5CY core is responsible to transfer a specific memory partition from the shared instruction memory to its local instruction memory. This is called the memory initialization stage, where each PE within the compute tile starts to move specific memory partitions from shared memory to local memory before application execution.

- 32-bit DTCM: In contrast to the ITCM, the DTCM is implemented as a single port on-chip BRAM/URAM with R/W interface to the RI5CY core D-Port. The DTCM is only accessed via its own coupled RI5CY core. Therefore, accessing local memory directly by other PEs is prevented and the local data memory has to be transferred to the shared data memory to be accessible by other PEs in the 32-bit tile. The DTCM is larger than the ITCM in terms of BRAM/URAM blocks as it hosts local PE data. During the memory initialization stage, the RI5CY core is responsible to load the local data from a specific partition from the shared memory to the DTCM prior to application execution. DTCM and ITCM are size configurable during design time based on target application requirements.

- I/D-Bridges: To allow seamless integration of PEs in 32-bit compute tile, the I/D-Ports of the RI5CY core are extended to be compatible with AXI-4 and AXI-Stream standards by implementing (D-, I-Bridges) as shown in Figure 3.2 (b). D-, I-Bridges allow the communication between the RI5CY core and the tile's memory-mapped peripherals through the shared AXI interconnect. Since the RI5CY core or the PE is the master unit on the proposed system. The PE AXI interfaces are master interfaces that permit a connection to any AXI slave peripherals inside the tile. As shown in Figure 3.2, the D-Bridge handles the RI5CY read/write memory requests (*req_D*) and the write-enable (*we*) signals from the D-Port interface by rerouting them based on the memory-mapped address range to the corresponding memory-mapped component (as shown in the bottom table in Figure 3.2). Hence, a finite state machine is implemented with seven states covering the read/write states to the (AXI interconnect, AXIS, ITCM_write and DTCM) interfaces based on address range selector. The address range selector specified which memory-mapped peripheral is requested by the RI5CY core based on the requested address range. According to the state and the address-range input, the D-Port interfaces (*data_write/read_D*, *valid_D*, *grant_D*) are re-connected to the corresponding interfaces. Similar to the D-Bridge, the I-Bridge is implemented as shown

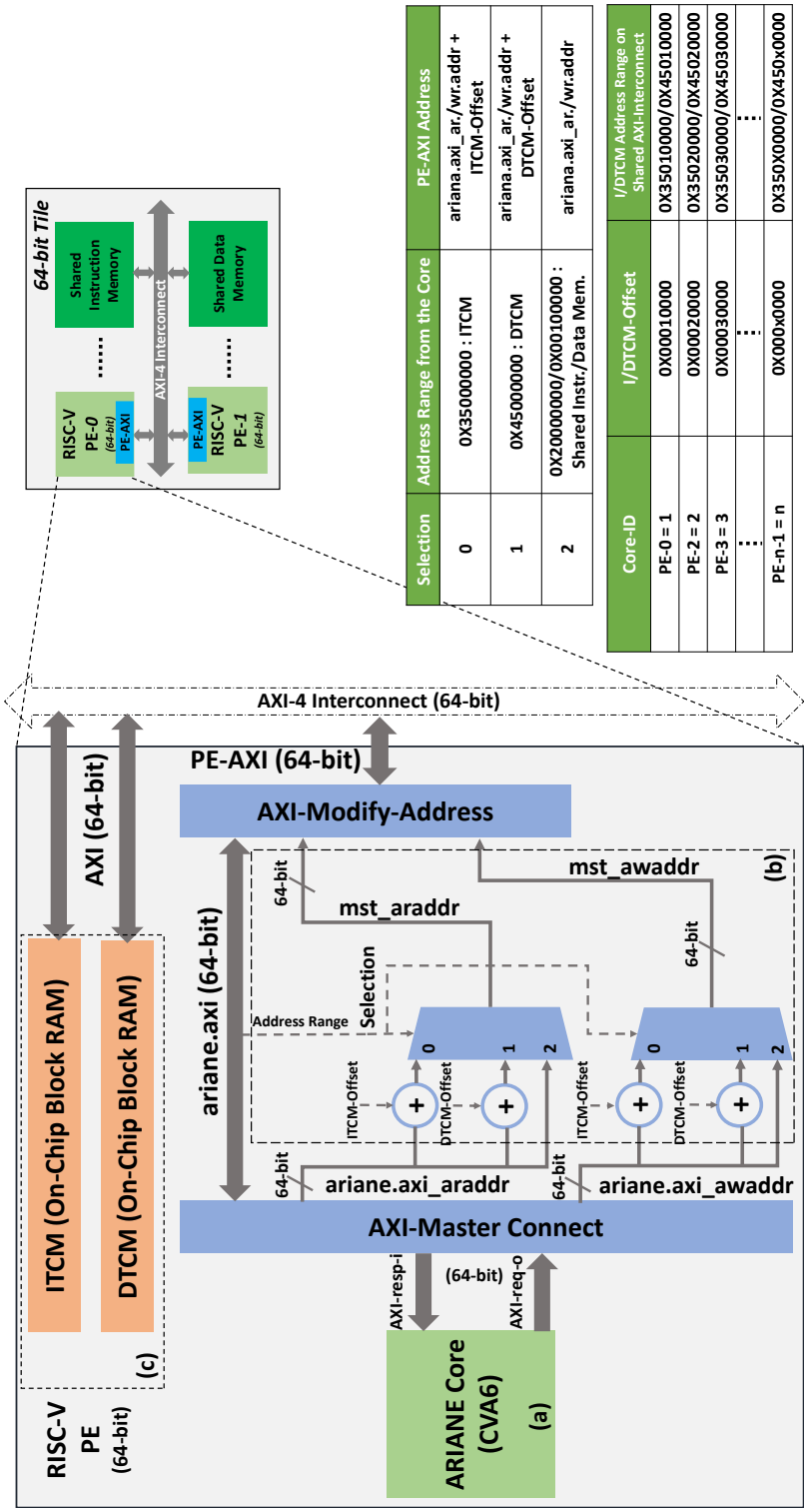


Figure 3.3: Schematic of 64-bit RISC-V based PE showing: (a) open-source RV64IMAC (CVA6/ARIANE) core, (b) address converter to access I/D TCM through the main AXI-4 interconnect, (c) on-chip I/D TCM and their connection to the CVA6 core through the main AXI-4 interconnect.

in Figure 3.2 (b) with a two states FSM for only reading from the ITCM or the shared instruction memory for instruction fetching. The address range selector specified which memory-mapped peripheral is requested (ITCM or the shared instruction memory) by the RI5CY core based on the requested address range. According to the state and the address-range input, the I-Port interfaces (*data_read_I*, *valid_I*, *grant_I*) are re-connected to the corresponding interfaces.

64-bit PE

The 64-bit PE consists of a single open-source Ariana (CVA6) soft-core processor [45] with an implemented local tightly coupled memory subsystem for data and instructions as shown in Figure 3.3. Ariane core is a 64-bit 6-stage pipeline in-order processor. The used core version in this work is configured to fully implement RV64IMAC [78]. Similar to the 32-bit PE, the tightly coupled memory subsystem is implemented using on-chip BRAM/URAM blocks as shown in Figure 3.3 (c). All memory blocks have a fixed word size of 64-bit compatible with RV64 ISA. Besides the Ariane core, the PE local memory subsystem and external interfaces are described as follows.

- 64-bit ITCM: The ITCM is implemented as a single port memory with R/W interface directly connected to the tile main AXI-interconnect. Based on the requested instruction memory address from the Ariane core, instructions can be fetched from the ITCM through the AXI interconnect directly to the core. During the memory initialization stage, the Ariane core is responsible to transfer a specific memory partition from the shared instruction memory to its local instruction memory via AXI interconnect. All PEs within the compute tile starts to move specific memory partitions from shared memory to local memory before application execution.

- 64-bit DTCM: The DTCM is implemented with R/W interface using a single port memory. Similar to the ITCM, the DTCM is directly connected to the tile main AXI interconnect. Based on the requested memory address from the Ariane core, local data can be loaded from the DTCM through the AXI interconnect directly to the core. During the memory initialization stage, the Ariane core is responsible to transfer its specified local data based on the target application memory partitions from the shared data memory to its local data memory via AXI interconnect. All PEs within the compute tile starts to load their local data from shared memory to local memory before application execution.

- PE address converter: In contrast to the 32-bit PE, the used open-source Ariane core is equipped already with interfaces that are compatible with AXI 64-bit standard interfaces (*AXI_resp*, *AXI_req*) to access instruction and data memories as shown in Figure 3.3 (a). Therefore, the design of a 64-bit PE is quite simple and does not require implementing extra I-/D-bridges or converters to make native core I/D-Ports compatible with AXI standards. However, an AXI-Master connect is implemented to re-route AXI request and response signals to different memory-mapped slaves peripherals (i.e. I/D-TCM, shared memory, and tile peripherals) as shown in Figure 3.3. Accordingly, D-/I-TCM are directly connected and accessed through the main tile AXI-interconnect in order to reduce the number of crossbars interconnects inside the PE and also memory access latency. Similarly to 32-bit PE, the local memory subsystem per PE is only accessed by its core. Therefore for each PE, I-/D-TCM_offsets are inserted to core's *AXI-resp/-req* R/W addresses signals (as shown in Figure 3.3 (b)) to modify the AXI addresses sent to the shared AXI-interconnect so that each core can access its local memory. As mentioned in

the right tables in Figure 3.3, I-/D-TCMs for all PEs have the same address range for all cores, but they have unique address ranges on the shared AXI-interconnect. Therefore, address modifications are required for each PE/core to access its corresponding I-/D-TCM. Accordingly, based on that implementation, we reduced the number of crossbar interconnects to be only the shared AXI interconnects which leads to a decrease in memory access latency (4 clock cycles) to local and shared memory.

3.2 System Scalability and Communication Model

Typical many-core and MPSoC architectures are considered as a suitable platform to run multi-tasks applications. Each task is mapped to one or more PEs or processing clusters based on the computation requirements. The tasks are connected via a directed data flow graph that defines the data flow and the execution period of each task for a specific application. In this section, the NoC configuration is presented based on a 2-D mesh topology as the communication fabric between compute tiles for the tile-based architecture. In addition, a unified network interface is developed to provide a generic interface for sending and receiving data between compute tiles (64-/32-bit tiles) and NoC routers. Moreover, a communication model between compute tiles with unidirectional RX/TX channels is developed based on the NoC and NI architecture characteristics. The communication model applies a message-based communication approach initiated by the transmitting tile and ending by the receiving tile.

3.2.1 NoC Configuration and Unified Network Interface

A Network-on-Chip is used on large-scale Multi-Processor System-on-Chip or many-core architectures to connect dozens to hundreds of PEs or compute tiles together, providing an on-chip end-to-end communication paradigm and increasing the system scalability. In this dissertation, ARTNoC [11] a real-time NoC architecture is used for inter-tile communication in the proposed tile-based many-core architecture. The NoC provides guaranteed quality of service (QoS) in terms of data transfer bandwidth and end-to-end latency. In addition, the router architecture is highly modular and parametrizable. It supports different I/O port configurations, switching controls, buffering sizes and routing schemes.

The ARTNoC circuit-switched-based version is used in the implementation as it features a low area overhead compared to packet-switched-based NoC architectures. It allows better utilization of FPGA resources to provide a resource-efficient communication fabric for large size of compute tiles. In other words, the FPGA resources are not heavily used by the NoC architecture in order to free those resources to host tens of large compute tiles as proposed in Section 1. The NoC is based on a 2-D mesh topology with an XY-routing algorithm with configurable dimension size (e.g. 2x2, 3x3, etc.) and parametrized I/O data widths at design time. Furthermore, the NoC internal architecture consists of the following units:

- A five ports circuit-switched router including a control path circuitry and arbiters for path reservation,
- A crossbar to switch between the I/O ports and using a round-robin arbitration scheme,
- Multiple synchronous network links for communication between the routers as shown in Figure 3.1 (a).

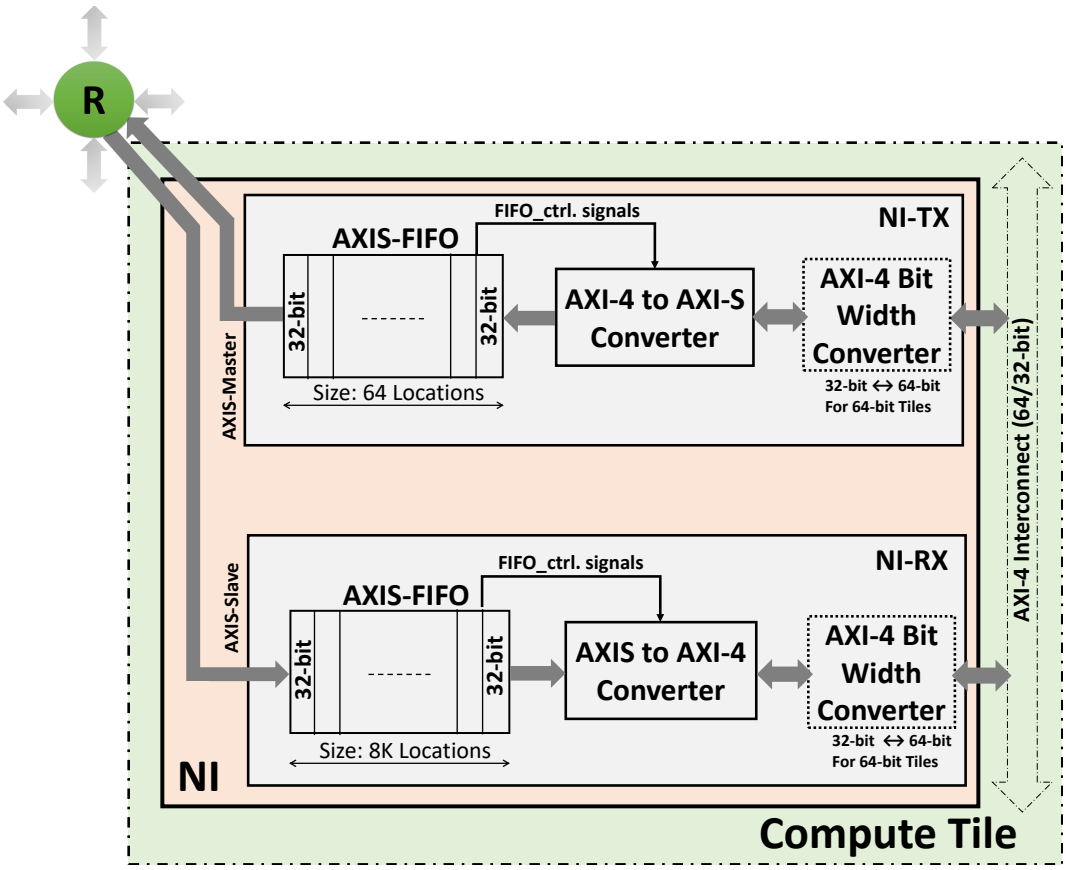


Figure 3.4: A unified network interface (NI) block diagram for many-core compute tiles.

The circuit-switched NoC reserves a static transmission path between the source and destination. This is performed by sending a single-flit setup packet from the source containing the X-Y coordinate of the destination node. Moreover, the NoC can transmit a single packet flit every single clock cycle with a 32-bit payload data.

In addition, a unified network interface (NI) is implemented to allow communication between compute tiles and the NoC. The unified NI is a memory mapped-peripherals that can be attached to the compute tile AXI interconnect. The NI is a generic hardware component that can be used by any compute tile in the tile-based architecture for both 64-bit tiles and 32-bit tiles. The NoC router I/O interfaces are compatible with the AXI-stream interface. Therefore, the proposed NI architecture is based on a flit-based streaming approach. Hence, the NI connects between the address-based shared bus used by the compute tile and the AXI-stream interface of the NoC router. An overview of the NI internal architecture is shown in Figure 3.4. The NI has two separate channels for sending and receiving data (NI-TX, NI-RX). It is connected to the AXI interconnect as AXI-slave memory-mapped peripheral that can be accessed by all compute tile PEs. The NI internal architecture consists of:

- An AXI-stream-based FIFO of 64 locations size in case of NI-TX and 8K locations in case of NI-RX to store/receive the transmitted and received packet flits from the NoC router.

- As the NoC is based on a 32-bit architecture, an AXI-stream to AXI-4 converter to connect the AXIS-FIFO and its control signals to the tile AXI-interconnect.

- An AXI-4 bit width converter is required for 64-bit compute tiles to convert between received 32-bit width and 64-bit to be compatible with tile data width.

A single PE can access the NI by setting a synchronization flag (for either sending or receiving) in the shared data memory indicating that the NI is blocked by this PE to prevent data interference by several NI requests from different PEs. The data flow between a PE to a NI is performed by setting a pointer to the data source address in the shared data memory to transmit a specified size of data. The data is transmitted in a form of a group of 32 packet-flits to the NI-TX FIFO. Similarly, in the receiving direction, the received packet flits are stored in the NI-RX AXIS-FIFO until a reading request comes from a certain PE to start data storing in the shared data memory. Transmitting and receiving data through NI can be done concurrently as the NI has two separate read/write channels to the AXI-interconnect.

3.2.2 Communication Model for Tile-based Architecture over the NoC

For communication and interaction between compute tiles, a message-based communication model is developed over the NoC to control data transfer between compute tiles based on running application requests. In other words, the communication model is considered as the network transport layer over the NoC hardware architecture to control data flow between compute tiles and maintain proper data transmission.

The interaction between compute tiles over the NoC is conducted through the message-based communication model. Where the compute tiles request to transmit and receive data to/from other compute tiles through a set of software modules. The communication model consists of software modules which are executed by any RISC-V core inside compute tiles to control and manage the NI peripheral to send or receive data over the NoC. Data are transferred through the NoC in a form of packets. All packets consist of 33 flits, where each flit is 32-bit. The first flit is the header flit which contains the destination address X_Y for the destination compute tile. The other 32 flits are the packet payload that contains transmitted or received data.

Figure 3.5 shows a sequence diagram of the developed communication model between compute tiles. The transmission is initiated by any PE in the source compute tile. The first transmitted packet flit is the header flit contains the X-Y coordinate of the tile destination followed by a stream of data packets equal to the size of the requested data transmission, where each packet contains 32 flits of payload data. The transmitted data is directly stored in the NI-RX AXIS-FIFO of the destination compute tile. Received packet flits are automatically stored in the AXIS-FIFO till the compute tile starts to store them on the shared data memory. Therefore, the receiving tile is not be blocked while the data is transmitted from the source compute tile and it can load the received data to shared data memory after a certain amount of time based on running tasks on the tile's PEs. However, the maximum amount of data that can be transmitted without blocking is equal to the size of NI-RX AXIS-FIFO which is set to 8K locations (32 KiB) for (8K×32-bit) data flits. In case the data is larger than 32 KiB, the source compute tile has to wait for an acknowledgement (ack) signal from the destination compute tile that the previous 8K data flits have been stored on the shared data memory and it is permitted to start sending another block of (8K×32-bit) data flits.

Accordingly, the proposed communication model guarantees no data loss during the transmission process. The communication model is realized through a set of software modules

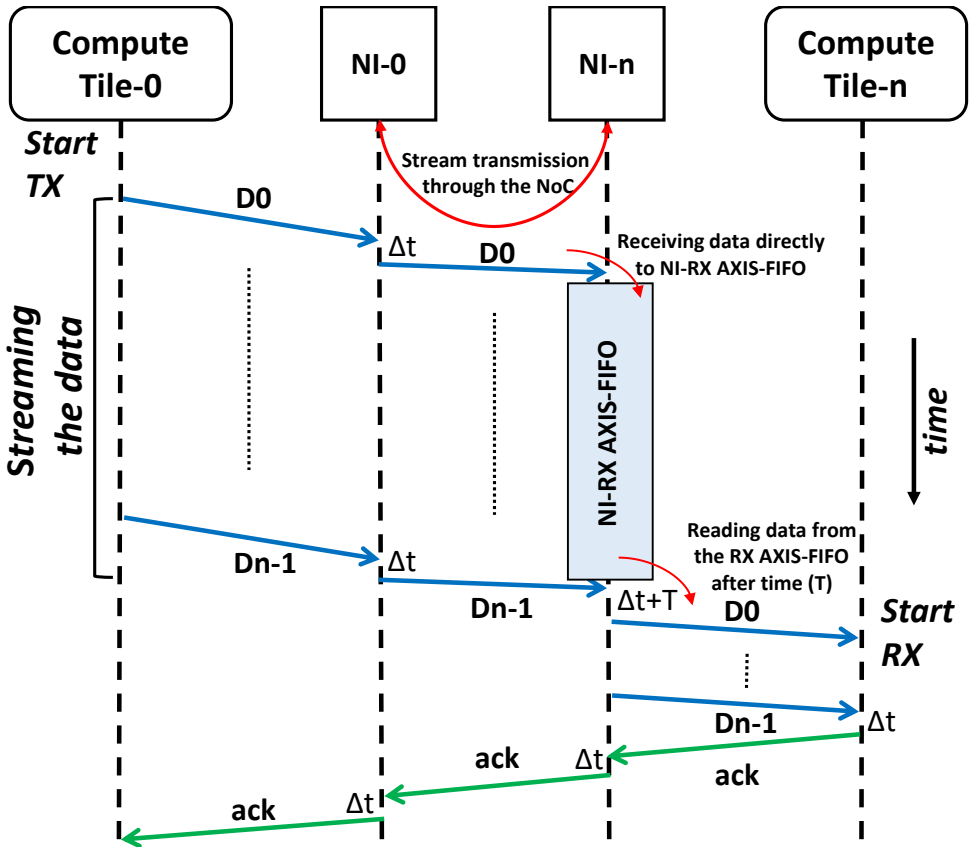


Figure 3.5: Sequence diagram of the message-based communication model between computing tiles over the NoC.

running on RISC-V PEs from any compute tiles to control and manage the NI to send and receive data. A detailed description of software modules is presented as follows:

Handling data transmission from compute tiles to the NoC

Listing 3.1 gives a detailed description of the communication software modules for data transmission via the NI-TX over the NoC. There are two data transmission software modules for the 32-bit and 64-bit compute tiles. For the 32-bit compute tile, 32-bit data are loaded from the shared data memory to the NI and then transmitted over the 32-bit NoC architecture. As shown in Listing 3.1 (line: 5), the header flit is transmitted including the destination compute tile followed by 32 flits of payload (line:6-7). The software module for data transmission transmits the data based on multiple data packets defined by *data_size*, where each transmission iteration sends a single packet to the NoC router and therefore to the destination compute tile. For 64-bit compute tile, as shown in Listing 3.1 (line:11), a 64-bit data flit has to be split into 2×32-bit data flits by the corresponding PE before transmission to the NI-TX (line:16-17). Therefore, a single transmitted data packet by a 64-bit tile contains 16x64-bit of data. Transmission software modules are executed from PEs ITCM as local functions for specific PE. To avoid multiple access of the NI-TX from different PEs, a synchronization flag is stored on the shared data memory to indicate the status of the NI-TX either active or idle. Accordingly, each PE is not allowed to access the NI-TX while it is active and used by other PE in the compute tile.

```
1 uint32_t*const NI_TX = (uint32_t)*0xA0000000; // NI_TX peripheral address
2 void send_data_32(uint_32t A[data_size], uint_32t data_size, uint_32t x_y_dest){ // 32-
    bit NI-TX module
3 uint32_t t = 0;
4     while(t < data_size/32){
5         NI_TX->data = x_y_dest;//Header flit transmission
6         for(int m = 0; m < 32; m++) // 32 flits data payload transmission
7             NI_TX->data = A[m+(32*t)];
8       }
9 return;
10 }
11 void send_data_64(uint_64t A[data_size], uint_32t data_size, uint_32t x_y_dest){ // 64-
    bit NI-TX module
12 uint32_t t = 0;
13     while(t < data_size/16){
14         NI_TX->data = x_y_dest;//Header flit transmission
15         for(int m = 0; m < 16; m++) // 32 flits data payload transmission
16             NI_TX->data = ((A[m+(16*t)]) & 0xffffffff); // lower 32-bit of the 64-bit
    sending data
17             NI_TX->data = ((A[m+(16*t)]) >> 32); // higher 32-bit of the 64-bit sending
    data
18       }
19 return;
20 }
```

Listing 3.1: NI data transmission software modules executed on RISC-V cores from general-purpose compute tiles.

Handling receiving data from the NoC to compute tiles

Listing 3.2 gives a detailed description of communication software modules for receiving data via the NI-RX over the NoC. There are two receiving data software modules for the 32-bit and 64-bit compute tiles. For the 32-bit compute tile, 32-bit data are received by the AXIS-FIFO to be stored later in the shared data memory. The AXIS-FIFO has a minimum size of 8K locations to store the transmitted data flits over the NoC. As shown in Listing 3.2 (line: 5), the PE checks for *FIFO_data_count* to ensure that there are available data flits to be received by the destination compute tile. Afterward, the received packet payload is extracted and stored in the shared data memory (line:6-7). The software module for receiving data receives the data based on multiple data packets defined by *data_size*, where each iteration reads a single packet from the AXIS-FIFO and stores it in the destination compute tile shared data memory. For 64-bit compute tile, as shown in Listing 3.2 (line:16), each received two data flits have to be concatenated again into one 64-bit data width before writing in shared data memory (line:22-24). Therefore, two received 32-bit data flits constitute 64-bit data for 64-bit compute tiles. Hence, data transmission between 64-bit compute tiles is based on data packets where each packet contains 16x64-bit data. After having received the complete transmitted data, the destination compute tile sends an acknowledgement data packet (line:10-12) to the source compute tile to confirm a successful receiving of all transmitted data as shown in Figure 3.5. Data receiving software modules are executed from PEs ITCM as local functions

```
1 uint32_t*const NI_RX = (uint32_t)*0xA000F000; // NI_RX peripheral address
2 uint32_t*const NI_TX = (uint32_t)*0xA0001000; // NI_TX peripheral address
3 void rec_data_32(uint_32t B[data_size], uint_32t data_size, uint_32t x_y_source){ // 32-
    bit NI-RX module
4   for(int t = 0; t < data_size/32; t++){
5        while(NI_RX->FIFO_data_count == 0); // waiting for data
6        for(int j = 0; j < 32; j++)
7           B[j+(32*t)] = NI_RX->data; // receiving 32 data flits
8       }
9       if(t == data_size/32){//send ack to the TX tile
10          NI_TX->data = x_y_source;
11          for(int j = 0; j < 32; j++)
12           NI_TX->data = 0x00000001;}//ack to the TX tile
13 return;}
14 void rec_data_64(uint_64t B[data_size], uint_32t data_size, uint_32t x_y_source){ // 64-
    bit NI-RX module
15  uint_32t lower_32_bit;
16  uint_32t higher_32_bit;
17  for(int t = 0; t < data_size/16; t++){
18       while(NI_RX->FIFO_data_count == 0); // waiting for data
19       for(int j = 0; j < 16; j++) // receiving 32 data flits, 16 x (2x32-bit)
20          lower_32_bit = NI_RX->data; // receiving lower 32-bit of the 64-bit transmitted
     data
21          higher_32_bit = NI_RX->data; // receiving higher 32-bit of the 64-bit
    transmitted data
22          B[j+(16*t)] = (uint64_t) higher_32_bit << 32 | lower_32_bit; // storing the
    received 64-bit data in the shared data memory
23      }
24      if(t == data_size/16){//send ack to the TX node
25          NI_TX->data = x_y_source;
26          for(int j = 0; j < 16; j++)
27           NI_TX->data = ((0x00000001 & 0xffffffff));//lower 32-bit of a 64-bit ack
    payload
28           NI_TX->data = ((0x00000000 >> 32));}//higher 32-bit of a 64-bit ack payload
29 return;}
```

Listing 3.2: NI data receiving software modules executed on RISC-V cores from general-purpose compute tile.

for specific PE. To avoid multiple access of the NI-RX from different PEs, a synchronization flag is stored on the shared data memory to indicate the status of the NI-RX as active or idle. Accordingly, each PE is not allowed to access the NI-RX while it is active and used by other PE in the compute tile.

3.3 Programming Method and Software Execution

In this dissertation, a bare-metal parallel programming method is developed for the proposed tile-based many-core architecture to generate multiple binary files from multi-tasks applica-

tions to be executed on many-core compute tiles. Each compute tile executes a separate binary file for its mapped task, we consider static task mapping over the selected number and type of compute tile that is conducted by the programmer/user prior to application execution. Each compute tile is programmed individually from other compute tiles and the shared instruction memory is used as a boot memory for each compute tile. As the proposed tile-based architecture supports single-core and multi-core tile architecture, the programming method and tasks execution are developed to support parallel programming over a multi-core architecture. As a general approach, shared memory is used to exchange data between PEs. Accordingly, synchronization between PEs which is essential for multi-core architectures can be achieved. Memory partitioning is an essential step in the development of the programming method. It defines memory sectors for local and shared data/instructions for each PE in the compute tile.

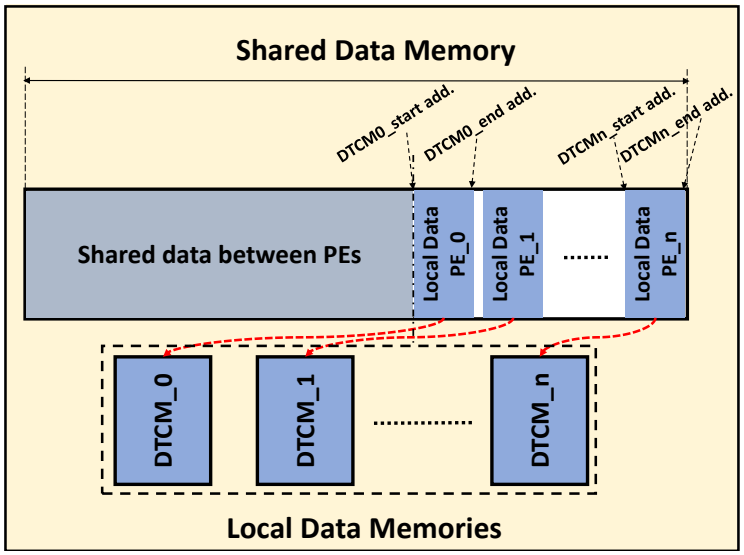
Before application execution, each PE executes its mapped software kernels from a specific location in shared or local instruction memories. Also, data memory locations for each PE are defined and determined prior to application execution. The compute tile shared data and instruction memories are used to store both shared and local data/instructions prior to execution. Afterward, during the memory initialization stage, each PE starts to load its local data/instruction to its local memories. Accordingly, the programming method is based on three steps as follows:

- The first step is the generation of separate binary files for each compute tile.
- The second step is the memory partitioning to define memory executable sectors on shared and local memories.
- The third step is the memory initialization stage executed on each compute tile separately.

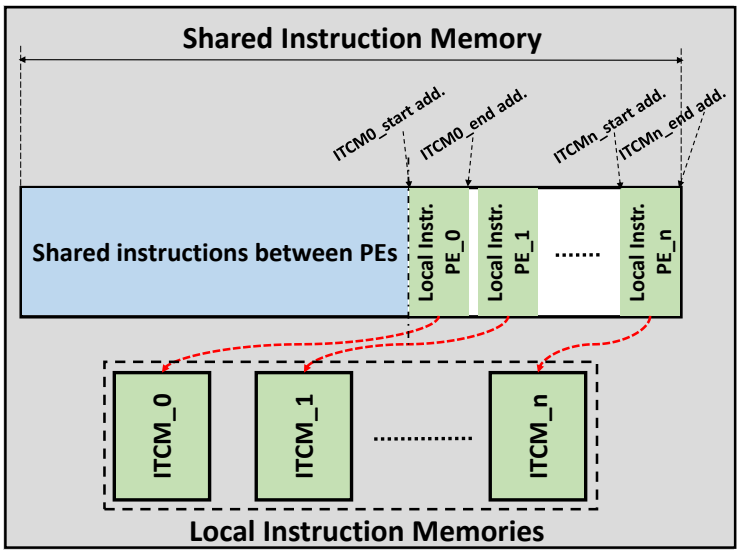
Figure 3.6 shows a schematic of memory partitioning to define memory sectors for each compute tiles. Initially, the shared data memory is loaded with local data for each PE in the compute tile as shown in Figure 3.6 (a). Each PE has a specific sector in the shared data memory during initialization. Memory sectors for local data are not overlapped with the shared data sector. For each PE memory sector, a start address (*DTCM_start_add*) and end address (*DTCM_end_add*) are defined by the linker script. During the memory initialization stage, each PE loads its local data from the specific address defined by the linker script to its local data memory.

Similarly, the shared instruction memory is loaded with local instruction for each PE in the compute tile as shown in Figure 3.6 (b). Each PE has a specific sector in the shared instruction memory during the initial loading of the executable binary file. Memory sectors for local instructions are not overlapped with the shared instructions sector. For each PE memory sector, a start address (*ITCM_start_add*) and end address (*ITCM_end_add*) are defined by the linker script. During the memory initialization stage, each PE loads its local instruction from the specific address defined by the linker script to its local instruction memory.

Listing 3.3 shows a detailed description of the developed linker script sections for a single compute tile memory mapping and partitioning. The linker script has several sections for ITCMs, DTCMs, and shared memory sectors. It describes the memory sectors shown in Figure 3.6 in terms of start, end addresses and memory partition sizes for both local and shared memories. The linker script is included to the toolchain and it is developed using the linker command language. The linker script consists of four sections described as follows:



(a) Data memory sectors.



(b) Instruction memory sectors.

Figure 3.6: Memory sectors of shared and local instruction and data memories for a single compute tile.

1. The *ENTRY* section is used to set the running application's entry point, it defines shared and local memories address space (memory-mapped addresses) and their sizes as shown in Listing 3.3 (line:1-12). Also, it set the starting address for the bootloader to load application instructions and data from either shared or local data/instructions memories.

2. The ITCM sections define the start, end addresses, and ITCM size on the shared instruction memory during the memory initialization stage (see Listing 3.3 (line:22-35)). ITCM sections specify local instruction memory attributed (*.ITCM_x*) for each PE ITCM to indicate that the local memory is the boot memory for specific application functions.

```
1   ENTRY(_start)
2   MEMORY
3   {
4   instr_mem(rx): ORIGIN =   0x00100000, LENGTH = 64k   /* shared instruction memory start address and size */
5   data_mem(rwx): ORIGIN =   0x20000000, LENGTH = 256k  /* shared data memory start address and size */
6   ITCM0(rw): ORIGIN = 0x35000000, LENGTH = 32k         /* loacal instruction memory start address and size for PE_0 */
7   ....
8   ITCM3(rw): ORIGIN = 0x35000000, LENGTH = 32k         /* local instruction memory start address and size for PE_0 */
9   DTCM0(rw): ORIGIN = 0x45000000, LENGTH = 64k         /* local data memory start address and size for PE_0 */
10  ....
11  DTCM3(rw): ORIGIN = 0x45000000, LENGTH = 64k         /* local data memory start address and size for PE_0 */
12  }
13  SECTIONS
14  {
15    __instr_mem_START = ORIGIN(instr_mem);
16    __instr_mem_SIZE  = LENGTH(instr_mem);
17    __data_mem_START = ORIGIN(data_mem);
18    __data_mem_SIZE  = LENGTH(data_mem);
19    __DTCM0_START = ORIGIN(DTCM0);
20    __DTCM0_LENGTH = LENGTH(DTCM0);
21    ....
22      /* ITCMs memory sectors */
23    .ITCM0_init_start : ALIGN(8)
24    {
25    __ITCM0_INIT_START = .;
26    } >instr_mem
27    .ITCM0 : ALIGN(8)
28    {
29    __ITCM0_START = .;
30  *(.ITCM_0);
31    } >ITCM0 AT>instr_mem
32    .ITCM0_init_end : ALIGN(8)
33    {
34    __ITCM0_INIT_END = .;
35    } > instr_mem
36      ....
37      /* DTCMs memory sectors*/
38    .DTCM0_init_start : ALIGN(8)
39    {
40    __DTCM0_INIT_START = .;
41    } >instr_mem
42    .DTCM0 : ALIGN(8)
43    {
44    __DTCM0_START = .;
45  *(.DTCM_0);
46    } >DTCM0 AT>instr_mem
47    .DTCM0_init_end : ALIGN(8)
48    {
49    __DTCM0_INIT_END = .;
50    } > instr_mem
51      ....
52      /*SHARED data_mem sectors*/
53    .bss : ALIGN(8)
54    {
55  *(.bss);
56    } > data_mem
57    .sbss : { *(.sbss); } > data_mem
58    .data_mem_init_start : ALIGN(8)
59    {
60    __data_mem_INIT_START = .;
61    } >instr_mem
62    .rodata : ALIGN(8)
63    {
64    __ram_START = .;
65  *(.rodata);
66    } > data_mem AT>instr_mem
67    .data : ALIGN(8)
68    {
69  *(.data);
70    } > data_mem AT>instr_mem
71    .data_mem_init_end : ALIGN(8)
72    {
73    __data_mem_INIT_END = .;
74    } > instr_mem
75  }
```

Listing 3.3: General-purpose compute tile linker script for single-core and multi-core architectures.

3. The DTCM sections define the start, end addresses, and DTCM size on the shared data memory during the memory initialization stage (see Listing 3.3 (line:38-50)). DTCM sections specify local data memory attributed (*.DTCM_x*) for each PE DTCM to indicate that the local memory is used as the data memory for specific application variables.

```
1 uint32_t * DTCM0 = (uint32_t*) &__DTCM0_START;
2 uint32_t * DTCM0_init = (uint32_t*) &__DTCM0_INIT_START;
3 uint32_t * ITCM0 = (uint32_t*) &__ITCM0_START;
4 uint32_t * ITCM0_init = (uint32_t*) &__ITCM0_INIT_START;
5 ....
6 uint32_t * DTCMx = (uint32_t*) &__DTCMx_START;
7 uint32_t * DTCMx_init = (uint32_t*) &__DTCMx_INIT_START;
8 uint32_t * ITCMx = (uint32_t*) &__ITCMx_START;
9 uint32_t * ITCMx_init = (uint32_t*) &__ITCMx_INIT_START;
10 uint32_t PE_id = read_csr(0xF14);
11 void init_main(){
12   if(PE_id == 0){
13     for(uint32_t i = 0; i <= &__DTCM0_INIT_END - &__DTCM0_INIT_START; i++)DTCM0[i] =
    DTCM0_init[i];
14     for(uint32_t i = 0; i <= &__ITCM0_INIT_END - &__ITCM0_INIT_START; i++)ITCM0[i] =
    ITCM0_init[i];
15     main_PE_0();}
16   ...
17   if(PE_id == x) {
18     for(uint32_t i = 0; i <= &__DTCMx_INIT_END - &__DTCMx_INIT_START; i++)DTCMx[i] =
    DTCMx_init[i];
19     for(uint32_t i = 0; i <= &__ITCMx_INIT_END - &__ITCMx_INIT_START; i++)ITCMx[i] =
    ITCMx_init[i];
20     main_PE_x();}}
```

Listing 3.4: Memory initialization stage (init.c) of a single general-purpose compute tile.

4. The shared data memory section defines the start, end addresses, and size of the shared sector in the shared data memory (see Listing 3.3 (line:53-75)). This section is used by static allocated variables (global variables) of the running applications. The read-only data input section (.rodata) Listing 3.3 (line:65) contains constant values. Global modifiable data are placed in the (.data) Listing 3.3 (line:69) section. All variables in this section could have pre-initialized values, so they need to be initialized during the program boot like the local data.

Listing 3.4 shows a detailed description of the memory initialization stage (*init.c*) for a single compute tile. The memory initialization stage is conducted before application execution from shared instruction memory where each PE loads its own local data/instructions from I/DTCM sectors on shared data/instruction memories to local memories. Based on PE number (*RISC-V core ID = CSR_MHARTID*), the memory initialization software module starts to move the local data/instruction from shared to local memories as shown in Listing 3.4 (line:13-14). Afterwards, it calls the corresponding PE main function (*main_PE_x*) to start application execution.

Listing 3.5 shows a sample software implementation over a single compute tile. Where each function which has to be executed from a local ITCM has to be preceded with a memory section attribute (*__attribute__(section(".itcm_0"))*) which defines its executable ITCM for a specific PE. Similarly, local data variables have to be preceded with memory section attribute (*__attribute__(section(".dtcm_0"))*) which defines its executable DTCM for a specific PE. Also, application parallelization is conducted by mapping parallel tasks over the PEs to be executed

```
1 #include "init.h"
2 __attribute__((section(".DTCM0")))
3 uint32_t dummy_0 [N];
4 __attribute__((section(".DTCMx")))
5 uint32_t dummy_x [M];
6 void main_PE_0(void){ ..... }
7 void main_PE_x(void){ ..... }
8 __attribute__((section(".ITCM0")))
9 void func_0(){ .....}
10 __attribute__((section(".ITCMx")))
11 void func_x(){ ..... }
```

Listing 3.5: A sample software implementation over a single multi-core compute tile.

within the PE main software function (*void main_PE_x(void)*) and using global variables as shared resources to be accessed by all PEs. In addition, any software functions which are not preceded by memory attributes are executed from the shared instruction memory and can be accessed by all PEs.

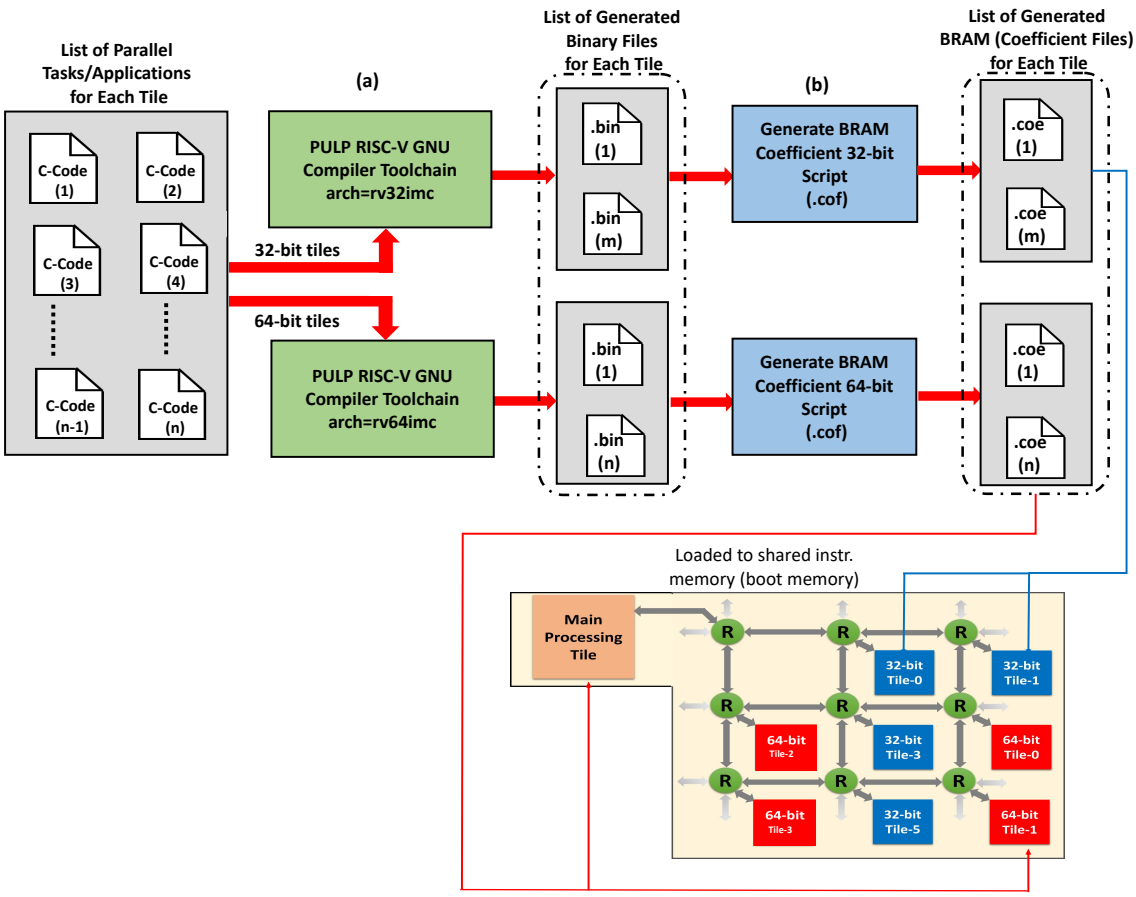


Figure 3.7: Schematic of the many-core programming flow including (a) building application tasks source codes targeting 32-/64-bit ISA, (b) generation of BRAM coefficient files to be stored on shared instruction memory (boot memory) of target compute tiles.

The PULP-RISC-V GNU toolchain [167] is used to compile C source codes for the 32-bit/64-bit compute tiles architecture. As shown in Figure 3.7, a list of generated binary files (.bin) for the 32-/64-bit compute tiles are the output of the compilation process, each compute tile has a single and separate binary file that will be executed on the shared instruction memory (boot memory) of its corresponding tile. Afterwards, the generated (.bin) files are converted to verilog memory files that contain the set of instructions for each tile. Then BRAM coefficient files (.coe) are generated to be loaded on shared instruction BRAM blocks for each tile as shown in Figure 3.7 (b). The BRAM coefficient files can be loaded to shared instruction memory during design tile prior to synthesizing process or after the generation of bitstreams by using update memory tool from Xilinx to only update BRAM contents of generated bitstreams.

3.4 Evaluation

Physical hardware implementation, system scalability, run-time reconfiguration and performance analysis results for the proposed tile-based many-core architecture are discussed and presented in this section. Xilinx Virtex Ultrascale+ XCVU9P [168] is the target FPGA for implementation and prototyping of the proposed tile-based many-core architecture. Also, Vivado Design Suite HLx 2019.1 [169] is used for RTL synthesis, simulation, place and routing, and full and partial bitstream generation. In this section, the tile-based many-core architecture is evaluated based on:

- Hardware resource utilization and power consumption for different compute tiles and heterogeneous PEs are described in Section 3.1.
- Memory bandwidth scalability using different memory hierarchies (local/shared memory). Also, compute tile scalability in terms of the number of PEs inside the compute tile and its impact on the overall memory bandwidth of a single compute tile.
- System scalability and computing performance with respect to different numbers and types of compute tiles in terms of inter-tile data transfer latency, and computing operations per second (Op/s).

Benchmarks and test cases used for evaluation are written as software kernels over corresponding compute tiles using C programming language and compiled using the PULP-RISC-V GNU toolchain [167] as described in the previous section to generate the corresponding binary (.bin) files and coefficient files (.coe) to be loaded into the shared instruction memory of each compute tile.

The execution cycles used in this section are measured by the performance counter register (PCCR) of RV32/RV64 cores. The number of cycles measured by the PCCR can be read using (*read_csr assembly*) function called in the corresponding benchmark or test case kernels and stored back in the compute tile shared data memory. As shown in Figure 3.1 (Section 3.1), sd-card and UART peripherals are only accessed by the main processing tile which manages external data transfer between compute tiles and external peripherals. Therefore, for purposes of testing and evaluation each compute tile transmits evaluation results to the main processing tile to be transmitted to the UART peripheral.

3.4.1 Hardware Resource Usage and Prototyping

The tile-based many-core architecture has been developed and implemented using a modular and hierarchical design approach. Where architectural modules (i.e. RISC-V cores, PEs, memory blocks, interconnects, compute tiles, etc.) are implemented as intellectual property (IP) components to be integrated together to build heterogeneous compute tile modules for several many-core configurations. Inter-tile communication is implemented using the ARTNoC framework [11] to generate multiple 2-D mesh NoC dimensions based on selected many-core configuration sizes. The NoC module is implemented as a single parameterized module including routers and network links to create the selected 2-D mesh topology size for each many-core configuration. The NoC is being configured only during design-time based on 2-D mesh size, and the number of data flits per packet. The NoC supports stream transmission of data flits of 32-bit data width each.

In addition, dual-ported BRAM/URAM blocks used for instruction and shared memory inside compute tiles are implemented using Xilinx BRAM/URAM memory generator blocks with AXI-BRAM controllers. Also, BRAM/URAM configurations are conducted during design time of compute tiles prior to synthesis and place and routing. All tile-based many-core configurations and their required modules are synthesized and implemented targeting Xilinx Virtex Ultrascale+ XCVU9P FPGA. The proposed tile-based architecture is running using a single clock domain of 120 MHz for all implemented components and modules including the NoC and all compute tiles.

32-bit Tile Resource Utilization

Table 3.1 shows the hardware resource utilization of the 32-bit compute tile used for the proposed tile-based many-core architecture as depicted in Figure 3.1 (a). The 32-bit compute tile consists of:

1. 4xRV32-PEs with 4KiB ITCM and 16KiB DTCM for each PE.
2. 64 KiB shared instruction memory.
3. 256 KiB shared data memory.
4. Two NI for transmission and receiving (NI-TX, NI-RX).
5. A 32-bit AXI-4 interconnect.

Each 32-bit RV32-PE consumes (~0.6%) of the total target FPGA LUT, where the 32-bit compute tile consumes (2.6%) of the total target FPGA LUT mostly consumed by the 4 RV32-PEs. On-chip memory usage is distributed between BRAM and URAM blocks with a total percentage utilization of (~3%) for the 32-bit complete tile. As shared and local data memories are implemented using URAM blocks while all instruction memories are implemented using BRAM blocks for balanced on-chip memory utilization. In addition, the 32-bit compute tile is equipped with two NI channels for transmitting and receiving over the NoC, for NI-RX 15 BRAM blocks are used to implement the (8K×32-bit) AXIS-FIFO and single BRAM block for NI-TX AXIS-FIFO. The power consumption per single 32-bit compute tile is estimated by Vivado power estimation tool and it is equal to 0.562 W.

Table 3.1: Hardware resource utilization and power consumption of the 32-bit general-purpose compute tile (RV32-tile) targeting a Xilinx Virtex Ultrascale+ (XCVU9P) FPGA.

Compute Tiles and Modules			Resource Utilization					Estimated Power
			LUTs	FFs	BRAMs	URAMs	DSPs	
RV32 Tile (4-PEs)	Total (4-PEs)		30717 (2.6%)	13137 (0.55%)	36 (1.6%)	12 (1.25%)	24 (0.35%)	0.562 W @120 MHz
	Shared Instr. Mem. (64 KiB)		87	15	16	0	0	
	Shared data Mem. (256 KiB)		334	11	0	8	0	
	AXI-4 interconnect		1033	184	0	0	0	
	RV32 PE	RI5CY Core	6902 (0.58%)	2491 (0.1%)	0	0	6 (~0.09%)	
		Local ITCM (4 KiB)	49	34	1	0	0	
		Local DTCM (16 KiB)	134	71	0	1	0	
	NI-RX		563	1088	15	0	0	
	NI-TX		525	1123	1	0	0	

64-bit Tile Resource Utilization

Table 3.2 shows the hardware resource utilization of 64-bit compute tiles (w/single-PE, and w/2-PEs) used for the proposed tile-based many-core architecture as depicted in Figure 3.1 (b). The 64-bit compute tile consists of:

1. Single/2xRV64-PEs with 4KiB ITCM and 16KiB DTCM for each PE.
2. 64 KiB shared instruction memory.
3. 256 KiB shared data memory.
4. Two NI for transmission and receiving (NI-TX, NI-RX).
5. A 64-bit AXI-4 interconnect.

Similarly, 64-bit compute tiles (w/single-PE, and w/2-PEs) consume the same on-chip memory resources as the RV32 compute tile. In contrast, one 64-bit RISC-V-based PE is 6x the size of a 32-bit RISC-V-based PE in terms of resource utilization. Therefore, the maximum number of PEs per 64-bit compute tile is two to keep resource utilization under a certain limit for 64-bit tiles. As shown in Table 3.2, the 64-bit (w/single-PE) compute tile is 1.5x and the

Table 3.2: Hardware resource utilization and power consumption of the two 64-bit general-purpose compute tiles (RV64(1-PE), RV64(2-PEs)) targeting a Xilinx Virtex Ultrascale+ (XCVU9P) FPGA.

Compute Tiles and Modules			Resource Utilization					Estimated Power
			LUTs	FFs	BRAMs	URAMs	DSPs	
64-bit Tile	Shared Instr. Mem. (64 KiB)		87	42	16	0	0	
	Shared data Mem. (256 KiB)		334	365	0	8	0	
	AXI-4 interconnect		2893	1983	0	0	0	
	RV64 PE	ARIANE Core	39693 (3.35%)	22472 (0.95%)	44 (~2%)	0	27 (0.4%)	
		Local ITCM (4 KiB)	335	359	8	0	0	
		Local DTCM (16 KiB)	367	362	16	0	0	
	NI-RX		1575	2310	15	0	0	
	NI-TX		1644	2345	1	0	0	
	64-bit Tile (1-PE)		46311 (3.9%)	32903 (1.39%)	76 (3.5%)	8 (0.83%)	27 (0.4%)	0.819 W @120 MHz
	64-bit Tile (2-PEs)		95636 (8%)	68806 (2.9%)	168 (7%)	8 (0.83%)	54 (0.79%)	1.423 W @120 MHz

64-bit compute tile with two PEs is 3x the size of the RV32 compute tile respectively. Each 64-bit RV64-PE consumes (~3.34%) of the total target FPGA LUT, where the 64-bit compute tile (w/single-PE, and w/2-PEs) consumes (3.9%, 8%) of the total target FPGA LUT mostly consumed by the 64-bit RISC-V-based PEs. On-chip memory usage is distributed between BRAM and URAM blocks with a total percentage utilization of (~4%), (~8%) for the 64-bit compute tile with single and dual RV64 PE respectively. As shared and local data memories are implemented using URAM blocks while all instruction memories are implemented using BRAM blocks for balanced on-chip memory utilization. In addition, 64-bit compute tiles are equipped with two NI channels for transmitting and receiving over the NoC, for NI-RX 15 BRAM blocks are used to implement the (8K×32-bit) AXIS-FIFO and single BRAM block for NI-TX AXIS-FIFO. The power consumption is estimated by Vivado power estimation tool and it is equal to 0.819 W for a single 64-bit compute tile with one PE and equal to 1.423 W for a single 64-bit compute tile with dual PEs.

Table 3.3: Hardware resource utilization and power consumption of the main processing tiles (RV64(4-PEs)) targeting a Xilinx Virtex Ultrascale+ (XCVU9P) FPGA.

Compute Tiles and Modules			Resource Utilization					Estimated Power
			LUTs	FFs	BRAMs	URAMs	DSPs	
Main Processing Tile	Total (4-PEs)		218773 (18.5%)	131639 (5.56%)	348 (16.1%)	0 (0%)	114 (1.6%)	5.881 W @120 MHz
	Shared Instr. Mem. (64 KiB)		95	44	16	0	0	
	AXI-4 interconnect		6428	7932	0	0	0	
	RV64 PE	ARIANE Core	39693 (3.35%)	22472 (0.95%)	44 (~2%)	0	27 (0.4%)	
		Local ITCM (4 KiB)	335	359	8	0	0	
		Local DTCM (16 KiB)	367	362	16	0	0	
	NI-RX		1575	2310	15	0	0	
	NI-TX		1644	2345	1	0	0	
	DDR, UART, SPI Ctrl.		12365	21493	26	0	6	

Main Processing Tile Resource Utilization

Table 3.3 shows the hardware resource utilization of the main processing tile used for the proposed tile-based many-core architecture as depicted in Figure 3.1 (c). The main processing tile consists of:

1. 4xRV64-PEs with 4KiB ITCM and 16KiB DTCM for each PE.
2. 64 KiB shared instruction memory.
3. DDR, UART, serial peripheral interface (SPI) controllers.
4. Two NIs for transmission and receiving (NI-TX, NI-RX).
5. A 64-bit AXI-4 interconnect.

As shown in Table 3.3, the main processing tile consumes (18.5%) of total target FPGA LUT and (~16%) of on-chip memory as it is configured with four 64-bit PE. However, the main processing tile can be configured with less PEs in order to reduce resource utilization for smaller FPGAs or more design space. On-chip memory usage is distributed between BRAM and URAM blocks with a total percentage utilization of (~16%). As shared and local data

Table 3.4: Hardware resource utilization of several tile-based many-core sizes and types targeting a Xilinx Virtex Ultrascale+ (XCVU9P) FPGA.

Tile-based Size	Resource Utilization				
	LUTs	FFs	BRAMs	URAMs	DSPs
8x32-bit Tiles	245736 (20.8%)	105096 (4.4%)	288 (12.8%)	96 (10%)	192 (2.8%)
8x64-bit Tiles (1-PEs)	370488 (31.2%)	263224 (11.12%)	608 (28%)	64 (6.64%)	216 (3.2%)
8x64-bit Tiles (2-PEs)	765088 (64%)	550448 (23.2%)	1344 (56%)	64 (6.64%)	432 (6.32%)
Single Main Processing Tile	218773 (18.5%)	131636 (5.56%)	348 (16.1%)	0	114 (1.6%)
NoC (3x3) Configuration	64127 (5.42%)	4752 (0.2%)	0	0	0

memories are implemented using URAM blocks while all instruction memories are implemented using BRAM blocks for balanced on-chip memory utilization. In addition, the main processing tile is equipped with two NI channels for transmitting and receiving over the NoC, for NI-RX 15 BRAM blocks are used to implement the (8K×32-bit) AXIS-FIFO and single BRAM block for NI-TX AXIS-FIFO. The power consumption is estimated by Vivado power estimation tool and it is equal to 5.881 W for a single main processing tile.

Three many-core configurations are generated for total resource evaluation purposes. The three configurations are based on 3x3 2-D mesh topology size to support 9xtiles (a single main processing tile and 8xcompute tiles). The three tile-based configurations are as follows:

- The first configuration consists of a single main processing tile and 8x32-bit compute tiles. Table 3.4 first row shows the total resource utilization of 8x32-bit compute tiles which consume (20.8%), (12.8%), (10%) of total target FPGA LUTs, BRAMs, URAMs respectively. The last row shows the resource utilization of the 3x3 NoC configuration which consumes (5.42%) of the target FPGA LUTs. Overall, with the main processing tile, the total resource utilization percentage is (∼45%), (∼29%), (∼10%) of the target FPGA LUTs, BRAMs, URAMs respectively

- The second configuration consists of a single main processing tile and 8x64-bit compute tiles (with a single 64-bit PE). Table 3.4 second row shows the total resource utilization of 8x64-bit compute tiles (1-PE) which consume (31.2%), (28%), (6.64%) of total target FPGA LUTs, BRAMs, URAMs respectively. Overall, with the main processing tile, the total resource utilization percentage is (∼55.12%), (∼44%), (∼6.64%) of the target FPGA LUTs, BRAMs, URAMs respectively

- The third configuration consists of a single main processing tile and 8x64-bit compute tiles (with a dual 64-bit PEs). Table 3.4 third row shows the total resource utilization of the 8x64-bit compute tiles (2-PEs) which consume (64%), (56%), (6.64%) of total target FPGA LUTs, BRAMs, URAMs respectively. Overall, with the main processing tile, the total resource utilization percentage is (~88%), (~72.1%), (~6.64%) of the target FPGA LUTs, BRAMs, URAMs respectively.

3.4.2 Memory Bandwidth Scalability

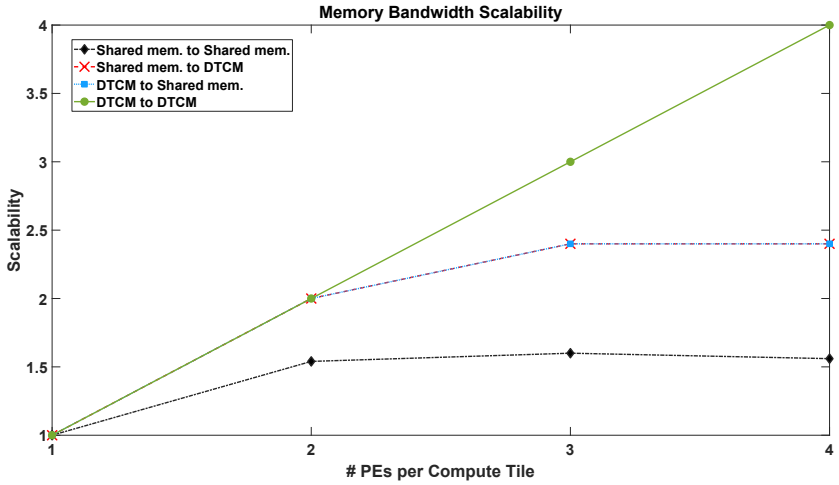
In this subsection, the tile-based many-core architecture is evaluated based on the achievable memory bandwidth in terms of data transfer rate between PEs and storage units (local and shared data memories) using different tile-based configurations. The evaluation is conducted over different numbers and types of compute tiles supported by the tile-based many-core architecture ranging from a single compute tile to eight compute tiles. Also, memory bandwidth scalability in proportion to the number of PEs per compute tile, as well as the scalable number of compute tiles, is evaluated.

Figure 3.8 shows the memory bandwidth scalability for a single compute tile (32-/64-bit). Both 32-bit and 64-bit tiles are using an AXI-4 Interconnect to communicate between the PEs and shared data memory. The memory bandwidth is measured by a parallel execution of a copy function on all PEs to copy a data size of 4 KiB through three evaluation scenarios:

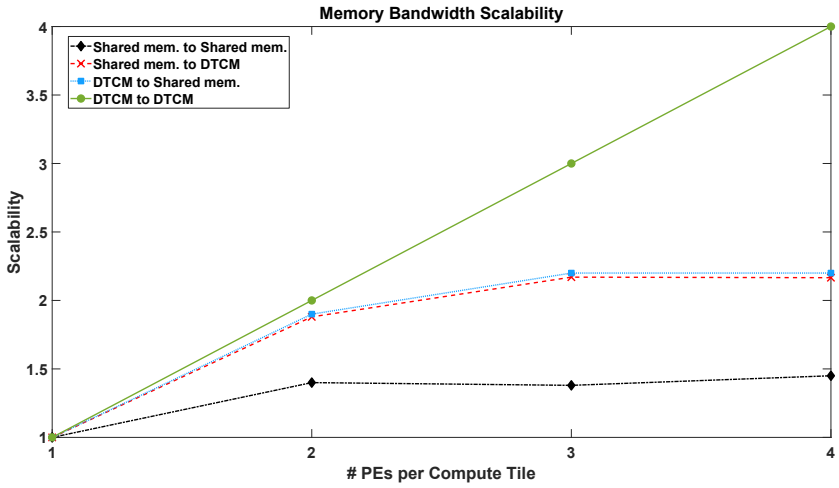
1. Shared data memory to shared data memory (SH-SH).
2. From shared data memory to the DTCM (SH-DTCM) or vice versa (DTCM-SH).
3. From DTCM to DTCM (DTCM-DTCM).

As a result, the data transfer bandwidth in case of shared to shared data memory is scaled by 1.5x using two PEs compared to one PE for both 32-bit and 64-bit compute tiles. However, if the dual-port data memory is used, the memory bandwidth is not scaled by the same factor due to the waiting cycles consumed for address collision mitigation if two PEs write or read from the same address at the same time. In contrast, splitting the memory write destinations by using DTCM in (SH-DTCM) scenario exploits the dual-ported memory feature by increasing the scalability to 2x in case of using two PEs. Moreover, in case of using shared data memory for reading or writing, increasing the number of PEs over 2 will not increase the memory bandwidth scalability proportionally to the number of PEs due to the traffic contention through the AXI-interconnect for both 32-bit (see Figure 3.8 (a)) and 64-bit tiles (see Figure 3.8) (b). On the other hand, as shown in Figure 3.8, memory bandwidth is proportionally scalable with the increasing number of PEs in the case of using DTCMs for writing and reading in a non-uniform memory access (NUMA) mode.

Therefore, for a 32-bit tile the number of PEs has been set to a maximum of four PEs per compute tile. In addition, the total resource utilization of four PEs as mentioned in the previous subsection is within a moderate range to support a scalable number of compute tiles using a large FPGA such as Virtex Ultrascale+ devices. However, in case of 64-bit tile, only the main processing tile has four 64-bit PEs due to limited FPGA resources to support four PEs per all 64-bit compute tiles. Therefore, for 64-bit compute tile, a maximum number of two PEs per tile is supported (64-bit tile w/single PE), (64-bit tile w/dual-PEs).



(a) Memory bandwidth scalability for a single 32-bit compute tile.



(b) Memory bandwidth scalability for a single 64-bit compute tile.

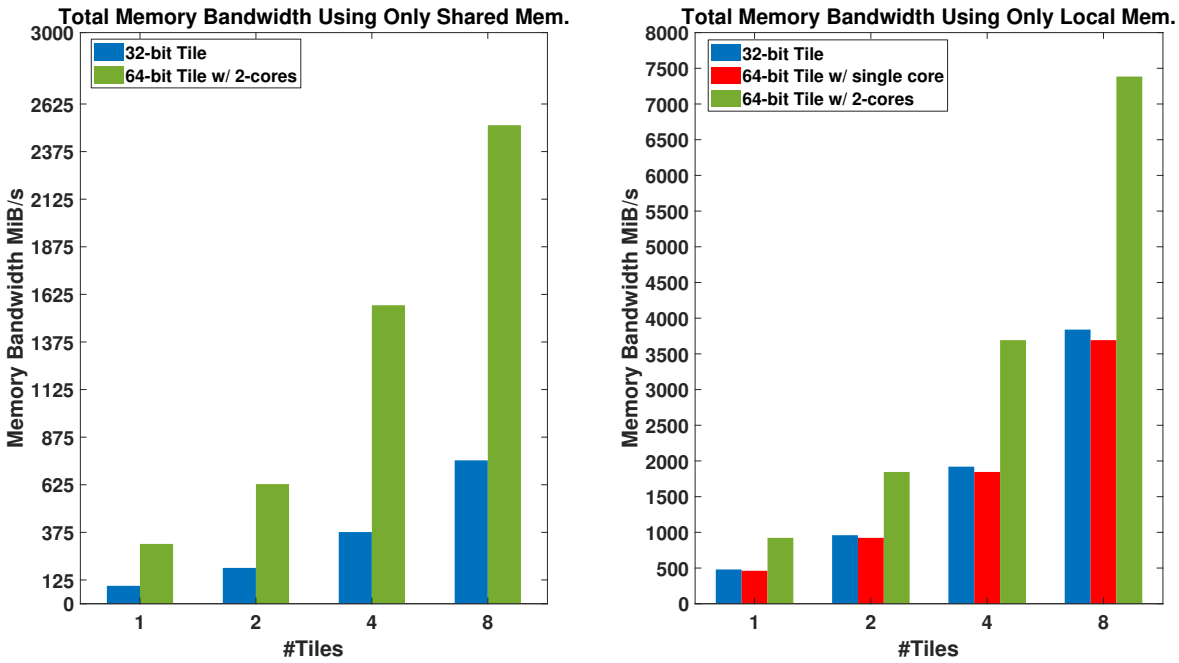
Figure 3.8: Memory bandwidth scalability for a single compute tile with respect to the number of RV32/64 cores per tile.

Similarly to memory scalability within a single tile, the total memory bandwidth for several tile-based many-core configurations is measured by parallel execution of several copy functions on all PEs inside compute tiles to copy a data size of 16 KiB through two evaluation scenarios:

1. By using only the shared data memory for multi-core based compute tiles (32-bit tiles, 64-bit tiles w/dual-PEs).

2. By using only the local data memory (DTCM) for all types of compute tiles (32-bit tiles, 64-bit tiles w/single-PE, 64-bit tile w/dual-PEs).

Several tile-based many-core configurations ranging from a single compute tile up to eight compute tiles are generated with different types of compute tiles (32-bit, 64-bit tiles). The



(a) Total memory bandwidth using only shared memory.

(b) Total memory bandwidth using only local memory.

Figure 3.9: Achievable memory bandwidth with respect to the number and types of many-core computing tiles using shared or local data memories at a clock frequency = 120 MHz (higher is better).

total memory bandwidth for the complete many-core architecture is calculated as follows:

$$Memory\ BW = \frac{freq. \times 2 \times data\_size}{n_{cycles}} \times n_{tile} \tag{3.1}$$

Where, *data_size* is the data transfer size from memory source to destination per bytes within one compute tile, the *data_size* is multiplied by 2 for simultaneous load and store operation using dual-ported memory, n_{cycles} is the measured number of clock cycles for memory data transfer latency per compute tile, and n_{tile} is the total number of compute tiles in the selected tile-based configuration. Figure 3.9 shows the memory bandwidth scalability up to eight compute tiles using different types of compute tiles with shared and local data memory scenarios. Memory bandwidth is approximately proportionally scalable with the increasing number of compute tiles for all types of supported compute tiles. For the shared memory scenario as shown in Figure 3.9 (a), using 64-bit tiles (2-PEs per tile) achieves a higher memory bandwidth of (~3.5x) compared to 32-bit tiles (4-PEs per tile). Therefore, supporting 64-bit memory transfer over 64-bit AXI interconnect for 64-bit tiles provides more memory bandwidth per tile and thus the total many-core configuration. Moreover, for 32-bit tiles, the memory bandwidth scalability is not increased proportionally by increasing the number of PEs due to the traffic contention through the AXI interconnect. As a result, for shared memory scenario, using 8x64-bit (w/2-PEs) tiles achieves a maximum memory bandwidth of (~2.5 GB/s) while 8x32-bit (w/4-PEs) achieves (~0.75 GB/s).

On the other hand, for the local memory scenario as shown in Figure 3.9 (b), the overall memory bandwidth scalability for all compute tiles is improved by (~4x) for 32-bit tiles and

by (~2.5x) for 64-bit (w/2-PEs) tiles compared to the shared memory scenario. Moreover, the memory bandwidth achieved by using 64-bit (w/single-PE) tiles is approximately the same achieved by 32-bit (w/4-PEs) tiles. Therefore, memory access latency is less in 64-bit tiles compared to 32-bit tiles due to fewer interconnects and data bridges usage inside 64-bit PEs between RISC-V cores and DTCMs. Consequently, 64-bit (w/2-PEs) tiles achieve (~2x) memory bandwidth compared to 64-bit (w/single-PE) tiles. As a result, the maximum achievable memory bandwidth using 8x64-bit (w/2-PEs) tiles is (7.4 GB/s) and (3.8 GB/s) for 8x32-bit tiles.

3.4.3 Computing Performance and Scalability

Design scalability determines the capability and flexibility of a parallel computing architecture to meet the required computing resources and communication data rate for parallel algorithms with growing complexity. Moreover, scalability is used to predict the performance of many-core architectures from the measured performance of single compute tiles. In this subsection, the tile-based many-core architecture is evaluated based on inter-tile data transfer latency, and computing performance in terms of integer operations per second (Op/s). The evaluation is conducted over different numbers and types of compute tiles supported by the tile-based many-core architecture.

Inter-Tile Data Transfer Latency Through the NoC

In order to measure inter-tile data transfer latency through the NoC. Two data transmission scenarios are evaluated for 32-bit and 64-bit tiles. Data transfer latency is measured from the transmitter compute tile (tile-TX) by measuring the time delay of transmitting a variant set of data sizes to the NI-RX of the receiving compute tile (tile-RX). The transmission time includes:

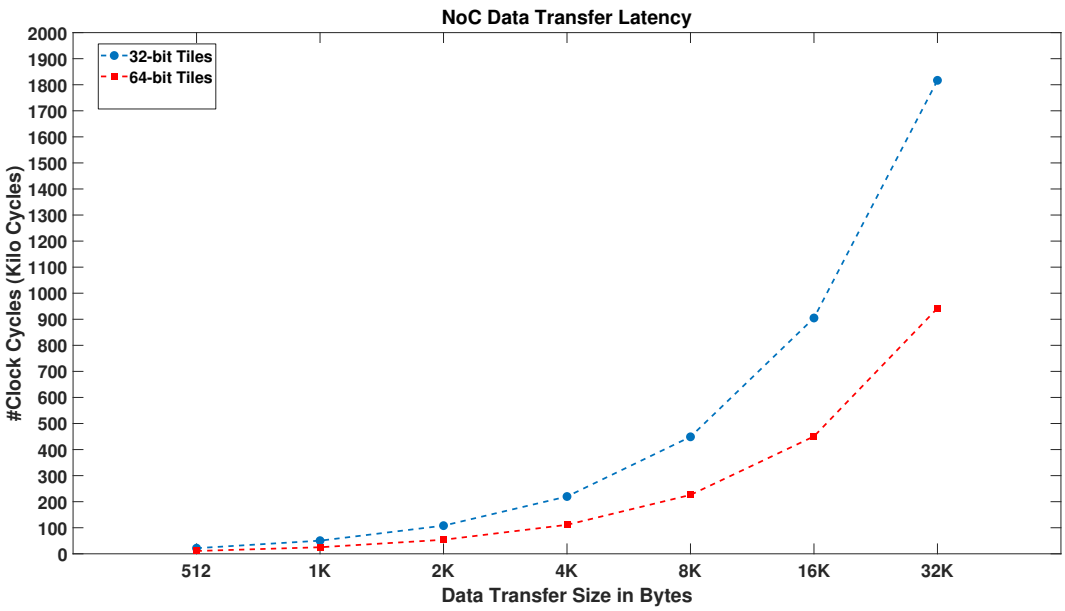


Figure 3.10: Data transfer latency over NoC between heterogeneous 32-/64-bit compute tiles.

1. The time overhead of loading the data from compute tile-TX shared data memory to NI-TX using the communication model software function running on a PE inside compute tile-TX.
2. The time overhead of transmitting the data from source to destination routers over the NoC.

Figure 3.10 shows the measured data transfer latency for both 32-bit and 64-bit tiles scenarios with up to 32KiB of data size. Since 64-bit tiles support 64-bit memory transfer from shared data memory to NI-TX compared to 32-bit memory transfer for 32-bit tiles, the data transfer latency from the 64-bit tile is (~ 2x) faster than the 32-bit tile. The transmission time overhead over the NoC is similar for both 32-bit and 64-bit tiles as the NoC supports stream data transmission of 32-bit. Therefore, transmission time between NoC source and destination routers is negligible compared to the time overhead required to load the data from shared data memory to the NI-TX inside the transmitting compute tile.

Computing Performance

In order to measure the computing performance of the tile-based many-core architecture, a parallel block matrix multiplication benchmark is implemented targeting different compute tiles for several many-core configurations. A fixed-point matrix multiplication benchmark is based on square matrix multiplication dimension for equal matrices partitioning for parallel execution over binary numbers of compute tiles. The parallel block matrix algorithm is used to partition matrix A into sub-matrices equal to the number of compute tiles. While matrix B is partitioned into sub-matrices equal to the number of PEs per compute tile. Each PE inside a compute tile computes the multiplication of a sub-matrix A with a sub-matrix B and stores the result in a sub-matrix C in shared data memory as shown in Figure 3.11.

32-/64-bit integer matrix multiplication algorithms are used over 32-bit and 64-bit compute

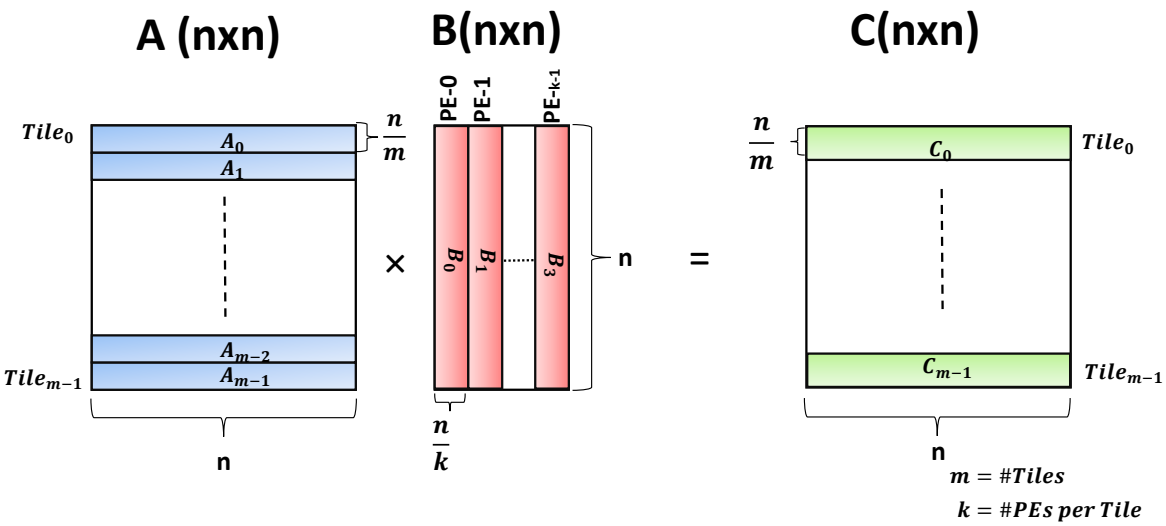


Figure 3.11: Block matrix multiplication partitioning over the tile-based many-core architecture.

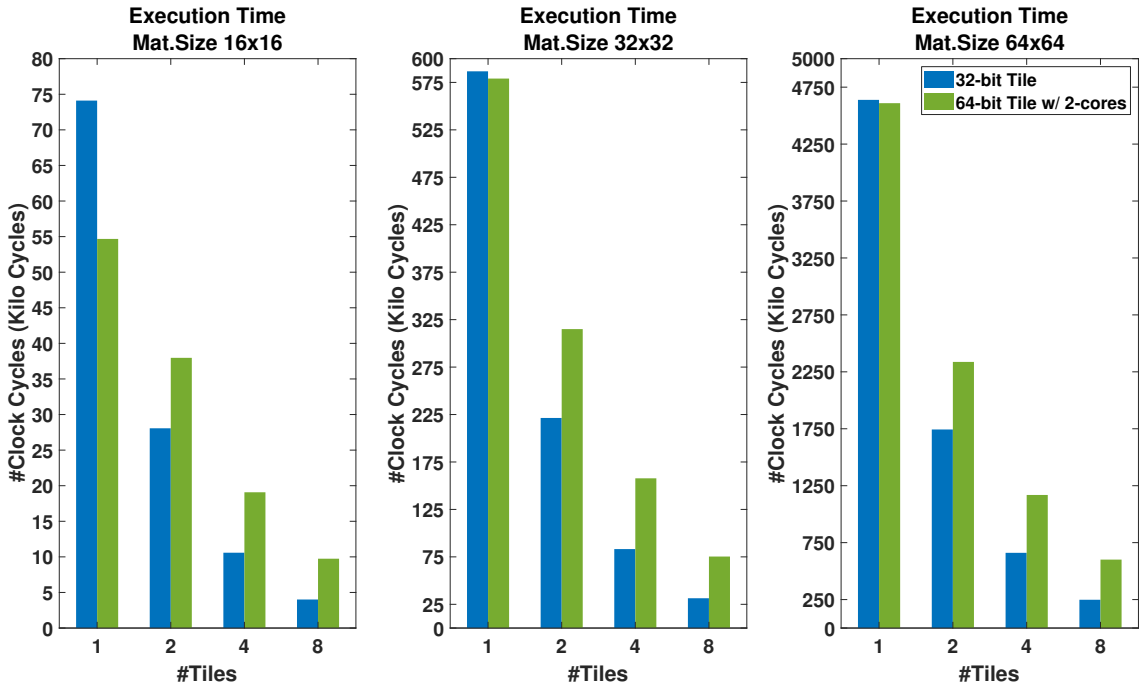


Figure 3.12: Execution time of matrix multiplication benchmark over different numbers and types of compute tiles using only compute tiles shared memory (lower is better).

tiles respectively. For evaluation, three many-core configurations are used with eight compute tiles. Each configuration supports only one type of compute tile (32-bit, 64-bit (w/single-PE), and 64-bit (w/2-PEs)). In addition, two evaluation scenarios are conducted based on selected memory hierarchy:

1. By using only the shared data memory to load and store matrices values for multi-core based compute tiles (32-bit tiles, 64-bit tiles w/2-PEs).
2. By using only the local data memory (DTCM) to load and store matrices values for all types of compute tiles (32-bit tiles, 64-bit tiles w/1-PEs, 64-bit tile w/2-PEs).

Figure 3.12 shows parallel matrix multiplication execution time for the two many-core configurations with only shared data memory using several types of compute tiles with three square matrix sizes (16x16, 32x32, 64x64). As a result, computing performance is proportionally scalable with an increasing number of compute tiles for 32-bit and 64-bit compute tiles using shared data memory. As shown in Figure 3.12, the computing performance of a single 32-bit compute tile is approximately the same as the 64-bit (w/2-PE) compute tile. Despite the higher number of PEs per 32-bit compute tile which increases computing performance, the 64-bit compute tile has a higher memory bandwidth compared to the 32-bit compute tile. Therefore, the required memory access time for loading and storing matrices values is less in the case of 64-bit compute tile which improves the overall computing performance with less number of PEs. However, by increasing the number of compute tiles, the many-core configuration with 32-bit compute tiles achieves less execution time by (~1.5x) compared to the many-core configuration with 64-bit compute tiles. That is due to the higher number of processing cores for 32-bit compute tiles compared to 64-bit compute tiles.

On the other hand, using local data memory will improve the computing performance by (~6x) for 32-bit compute tiles and (~3.5x) for 64-bit compute tiles as shown in Figure 3.13.

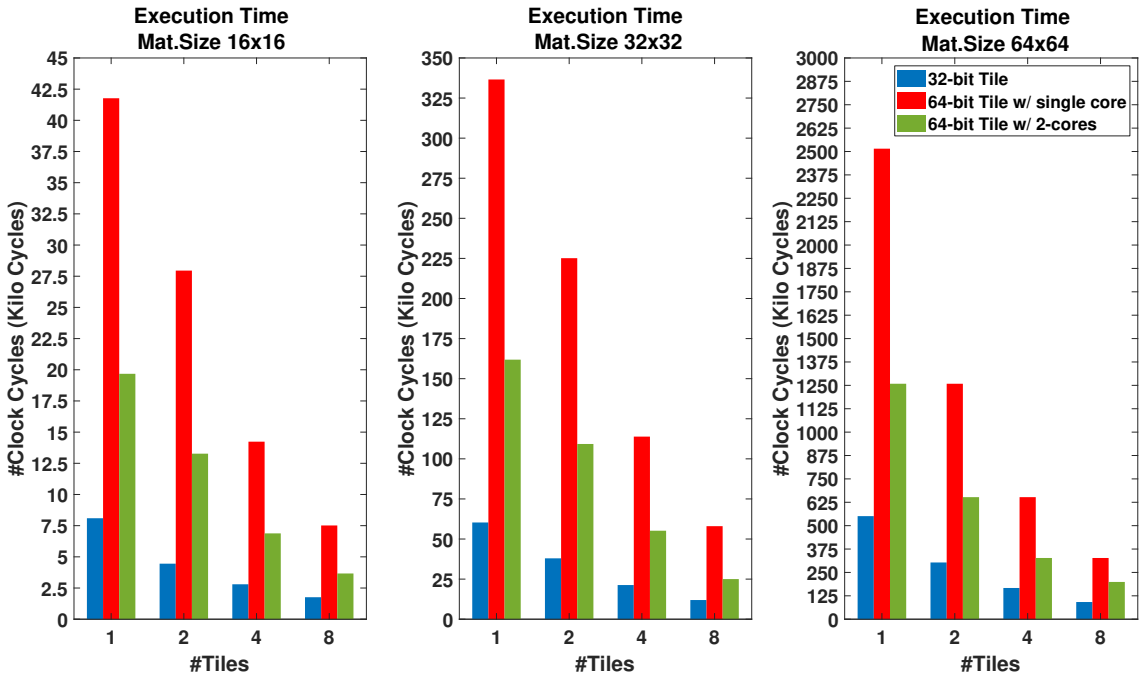


Figure 3.13: Execution time of matrix multiplication benchmark over different numbers and types of compute tiles using only compute tiles local memory (lower is better).

In local memory scenario, computing performance depends only on the number of PEs as each PE has its own data memory. Therefore, for all numbers of compute tiles, the 32-bit compute tile achieves less execution time by (~5x) in comparison with 64-bit compute tile (w/single-PEs). Furthermore, the total computing performance is calculated as follows:

$$Computing\ Performance\ (Op/s) = \frac{2 \times n^3}{n_{cycles}} \times freq. \tag{3.2}$$

Where n^3 is the computing complexity of square matrix multiplication of size (n×n), n_{cycles} is the execution time per clock cycles shown in Figure 3.12 and Figure 3.13, the multiplication by 2 is the number of multiply and accumulate (MAC) operations. Table 3.5 shows the computing performance in terms of the number of Op/s based on the matrix multiplication benchmark for the aforementioned tile-based many-core configurations and evaluation scenarios. As a result, the tile-based many-core architecture achieves a maximum 32-bit computing performance of (685 MOPS) configured with 8x32-bit (w/4-PEs) compute tiles using only local data memory. For 64-bit integer operations, a maximum computing performance of (316 MOPS) is achieved by 8x64-bit (w/2-PEs) tile-based configuration. In addition, the main processing tile achieves a maximum performance of (96 MOPS).

3.4.4 Comparison with State-of-the-Art

In this chapter, the goal is to provide a heterogeneous and adaptable tile-based many-core architecture to support seamless integration and communication between multiple RISC-V ISAs for realizing several many-core configurations. Our main contributions rely on the modularity and configurability of compute tiles to support variant requirements for compute and memory-bound applications. The proposed architecture specifically targets FPGA devices

Table 3.5: Computing performance for different numbers and types of compute tiles based on matrix multiplication benchmark at a clock frequency = 120 MHz.

Number and Types of Tiles		Performance	
		Only Shared Data Mem.	Only Local Data Mem. (DTCM)
Single Tile	RV32 (4-PEs)	14 MOPS	114 MOPS
	RV64 (1-PEs)	-	25 MOPS
	RV64 (2-PEs)	14 MOPS	50 MOPS
2xTiles	RV32 (4-PEs)	36 MOPS	207 MOPS
	RV64 (1-PEs)	-	52 MOPS
	RV64 (2-PEs)	27 MOPS	97 MOPS
4xTiles	RV32 (4-PEs)	96 MOPS	377 MOPS
	RV64 (1-PEs)	-	98 MOPS
	RV64 (2-PEs)	54 MOPS	193 MOPS
8xTiles	RV32 (4-PEs)	254 MOPS	685 MOPS
	RV64 (1-PEs)	-	192 MOPS
	RV64 (2-PEs)	105 MOPS	316 MOPS
Main Processing Tile	RV64 (4-PEs)	25 MOPS	96 MOPS

for fast prototyping and evaluation which make it suitable for design space exploration for many application domains. Several RISC-V based many-core architectures are previously proposed in the literature and as open-source platforms. However, design modularity and heterogeneity by supporting multiple ISAs are not supported by several state-of-the-art approaches. Besides, Table 3.6 shows a comparison between our proposed architecture and several state-of-the-art RISC-V-based many-core architectures targeting FPGA devices.

The comparison aims to evaluate hardware specifications and computing performance of our proposed architecture with other state-of-the-art approaches. Proposed architectures by [43], [159], and [170] are based on a single application class RISC-V core per tile supporting one ISA which increases the cost of scalability in terms of hardware resources required for interconnection for many-core realizations as well as high clock frequency to increase the compute performance of a single tile (in case of ESP [170]). In contrast, Andromeda architecture [158] combines several cores per tile adding more computing power to a single tile using lower clock frequency and reducing the cost of scalability. However, power consumption is the main bottleneck of using application class RISC-V processors in compute tiles, especially for high scalable many-core systems. Therefore, GRVIPhalanx [38] uses a simple RISC-V ISA (RV32I) to support tens of compute tiles with appropriate overall power consumption. However, computing performance is very limited as it only support RV32I. In comparison to them, our proposed architecture supports different types of compute tiles with more computing capabilities suitable for several application requirements. The RV32

Table 3.6: Comparison between state-of-the-art RISC-V based many-core architectures and the proposed modular and heterogeneous many-core architecture in terms of resources utilization and computing performance targeting FPGA platforms.

Architecture (# RV Cores/PEs per Tile, ISA) (Year)		Resource Utilization per Tile				Freq.	Power	Perf.	FPGA Device
		LUT	BRAM	URAM	DSP	(MHz)	per Tile	(8-Tiles)	
OpenPiton+Ariane [43] (Single-Core), RV64GC (2019)		90K	88	0	19	150	-	-	Virtex-Ultra.+ (XCVU9P)
Andromeda* [158] (4-Cores), RV64GC (2021)		131188	48	0	140	50	-	7.5 MOPS*	Synopsys HAPS (Virtex Ultra.)
Savas et al. [159] (Single-Core), RV64G (2020)		28241	258	0	15	113	-	-	Virtex-Ultra.+ (XCVU9P)
GRVIPhalanx [38] (8-Cores), RV32I (2016)		4800	12	0	0	150	0.34 W	-	Virtex-Ultra.+ (XCVU9P)
ESP [170] (Single-Core), RV64IMAC (2022)		50290	36	0	27	250	1.484 W	400 MOPS	Virtex-Ultra.+ (XCVU9P)
This Work (2022)	RV32-Tile (4-PEs), RV32IMC	30717	35	12	24	120	0.562 W	685 MOPS	Virtex-Ultra.+ (XCVU9P)
	RV64-Tile (Single-PE), RV64IMAC	46311	76	8	27	120	0.819 W	192 MOPS	
	RV64-Tile (2-PEs), RV64IMAC	95636	168	8	54	120	1.423 W	316 MOPS	

* Performance for Andromeda is reported by single precision floating point Op/s, the rest of architectures are measured by 32-/64-bit integer Op/s.

tile supports highly scalable many-core systems with less resource utilization and power consumption per tile. Also, RV64 tiles feature low resource utilization compared to [43], [158], and [170] supporting local and shared memory hierarchies per tile.

3.4.5 Use Cases Applications

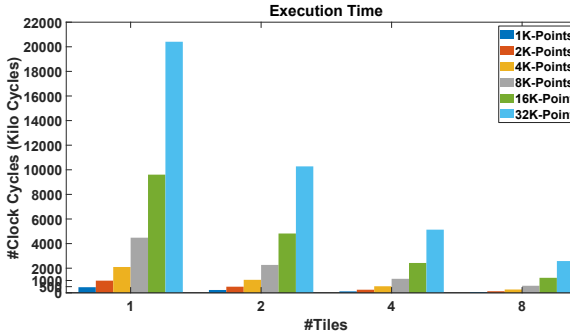
In this subsection, several signal processing and quantized neural network (QNN) kernels are used to evaluate the tile-based many-core architecture. The evaluation is based on total execution time and performance scalability. For all use cases applications in this subsection, only the shared data memory is used due to the large amount of data that needs to be loaded and stored during computation. Four signal processing kernels are considered for evaluation (FFT, square matrix inverse, 2-D convolution, and 3-D convolution) kernels. In addition, QNN inference kernels derived from the open-source PULP QNN library [171], [172]. In this subsection, three tile-based many-core configurations with 8xcompute tiles and a single main processing tile are used. Each configuration supports one homogeneous type of compute tiles (32-bit, 64-bit w/1-PE, 64-bit w/2-PEs) and the computation is only conducted over the homogeneous compute tiles. The main processing tiles is used to distribute input data among compute tiles (loaded from the external DDR) and collect output data through the NoC. The execution time does not include input and output transfer latency between compute tiles and the main processing tiles.

FFT kernels

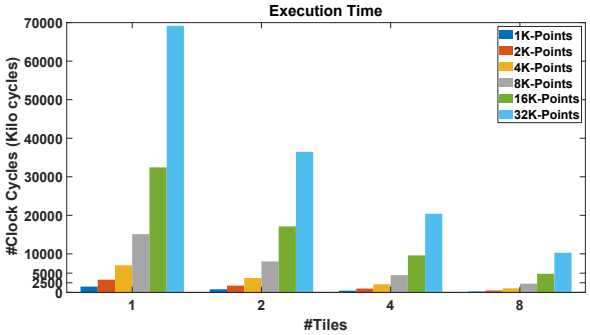
The Cooley-Tukey FFT algorithm is parallelly executed over several numbers of compute tiles (single compute tile up to eight compute tiles). Each compute tile receives the FFT input from the main processing tile and stores it in the shared data memory. The FFT input is equally distributed among the selected number of compute tiles. Next step, each PE inside the compute tile starts to execute a portion of the received input by the compute tiles. The workload is also equally distributed among the number of PEs within the compute tiles. Figure 3.14 shows the execution time of several FFT kernels with different N-point sizes from 1K to 32K. The FFT input and output data are 32-bit integer for the 32-bit compute tiles and 64-bit integer for 64-bit compute tiles. Figure 3.14 (a) shows execution time using only the many-core configuration with 32-bit compute tiles. Therefore, the FFT computing parallelization is conducted over 4xRV32PEs up to 32xRV32PEs. The computing scalability is approximately 2x by increasing the number of compute tiles with the same factors. On the other hand, as shown in Figure 3.14 (b, c), the execution time over 64-bit compute tiles are higher in comparison with 32-bit compute tiles as the number of PEs per tile is less and the FFT computation are conducted with 64-bit integer data. Similar to 32-bit tile-based configurations, the computing scalability is approximately 2x by increasing the number of 64-bit compute tiles with the same factors.

Matrix inverse kernels

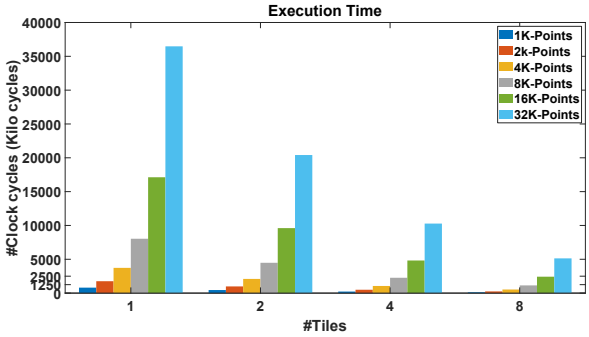
Matrix inverse is a fundamental matrix operation in signal processing applications. The inverse of a matrix has the property that when a matrix is multiplied by its inverse, the resulting matrix is the identity matrix. The inverse matrix kernels are implemented for only square-size matrices. The Gaussian-Jordan elimination algorithm is used to implement matrix inverse kernels. Gaussian-Jordan elimination calculates the inverse of an n x n matrix by



(a) FFT Kernels, 32-bit compute tiles (4-PEs).



(b) FFT Kernels, 64-bit compute tiles (1-PE).

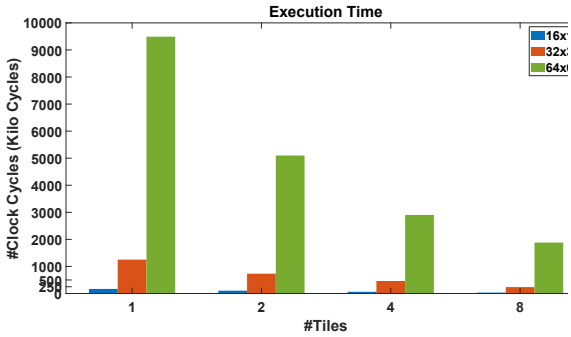


(c) FFT Kernels, 64-bit compute tiles (2-PEs).

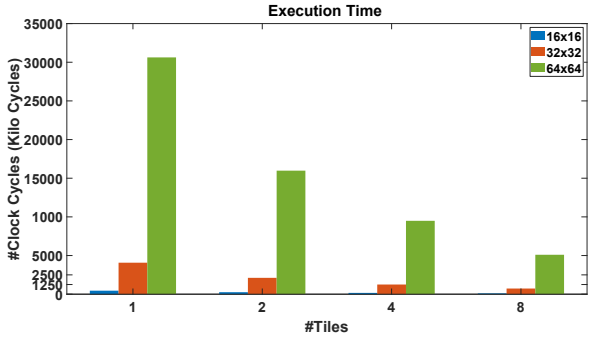
Figure 3.14: Execution time of several FFT kernels with different sizes over different numbers and types of compute tiles for multiple many-core configurations.

extending the matrix with an identity matrix of size n x n and doing the elementary row operations such that the left-hand side of the extended matrix becomes the identity matrix. In this case, the right-hand side of the extended matrix is the inverse of the input matrix. Parallelization of Gauss-Jordan elimination is conducted first by generating the identity matrix based on the input matrix size. Next step, the input matrix is equally row-wise partitioned over the selected number of compute tiles as well as over the PEs inside compute tiles. All input matrix partitions and identity matrix are stored on shared data memory per every compute tile. The row elimination process is conducted in parallel by all PEs independently to gain a speedup.

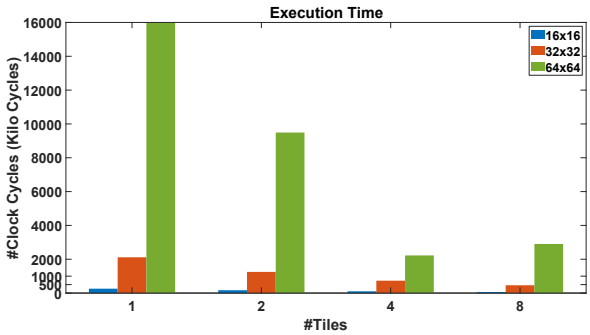
Figure 3.15 shows the execution time of several matrix inverse kernels with different input matrix sizes (16x16, 32x32, 64x64). The input matrix and the output inverse matrix element are integer 32-bit for the 32-bit compute tiles and 64-bit integer for 64-bit compute tiles. Figure 3.15 (a) shows execution time using only the many-core configuration with 32-bit compute tiles. Therefore, the matrix inverse computing parallelization is conducted over 4xRV32PEs up to 32xRV32PEs. The computing scalability is approximately 2x by increasing the number of compute tiles with the same factors. On the other hand, as shown in Figure 3.15 (b, c), the execution time over 64-bit compute tiles are higher in comparison with 32-bit compute tiles as the number of PEs per tile is less and the matrix inverse computation are conducted with 64-bit integer data. Similar to 32-bit tile-based configurations, the computing scalability is approximately 2x by increasing the number of 64-bit compute tiles with the same factors.



(a) Matrix inverse kernels, 32-bit compute tiles (4-PEs).



(b) Matrix inverse kernels, 64-bit compute tiles (1-PEs).



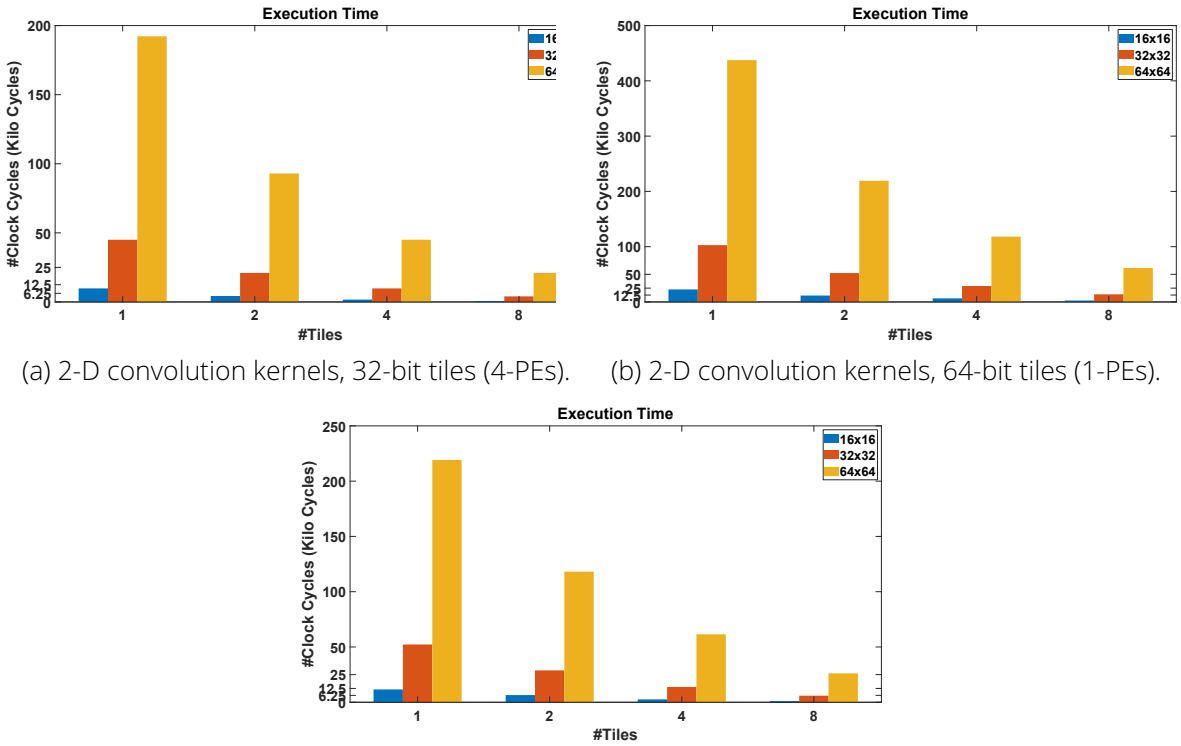
(c) Matrix inverse kernels, 64-bit compute tiles (2-PEs).

Figure 3.15: Execution time of several Matrix inverse kernels with different sizes over different numbers and types of compute tiles for multiple many-core configurations.

2-D convolution kernels

2-D parallel convolution kernels are considered for evaluation as they are commonly used for signal processing and neural network algorithms. 2-D parallel convolution kernels are evaluated over the aforementioned three many-core configurations based on 32-bit and 64-bit compute tiles. Also, 64-bit and 32-bit integer operations are supported by 32-bit and 64-bit compute tiles respectively. Parallel execution is conducted by partitioning the input matrix over the target number of compute tiles. In this use case, input matrices have square dimensions to be partitioned equally over a binary number of compute tiles. Inside each compute tile the input matrix is partitioned again over the number of PEs per tile. 2-D convolution is computed by a sliding window size of (3x3) across the assigned input matrix for each PE. Loading and storing operations from/to the memory during convolution are conducted using shared data memory for each PE due to the limited size of local data memory for larger sizes of input matrices to be executed.

Figure 3.16 shows the execution time of several 2-D convolution kernels with different input matrix sizes (16x16, 32x32, 64x64). Figure 3.16 (a) shows execution time using only the many-core configuration with 32-bit compute tiles. Therefore, the 2-D convolution parallelization is conducted over 4xRV32PEs up to 32xRV32PEs. The computing scalability is approximately 2x by increasing the number of compute tiles with the same factors. On the other hand, as shown in Figure 3.16 (b, c), the execution time over 64-bit compute tiles are higher in comparison with 32-bit compute tiles as the number of PEs per tile is less and the 2-D



(a) 2-D convolution kernels, 32-bit tiles (4-PEs). (b) 2-D convolution kernels, 64-bit tiles (1-PEs).

(c) 2-D convolution kernels, 64-bit tiles (2-PEs).

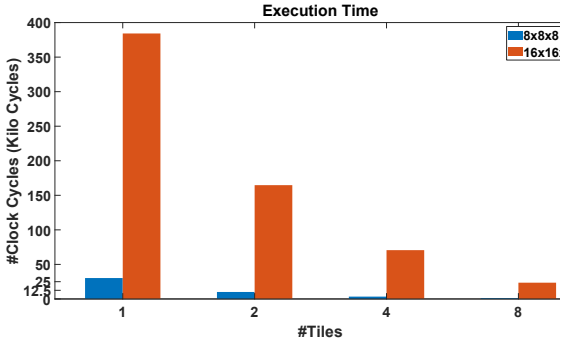
Figure 3.16: Execution time of several 2-D convolution kernels with different sizes over different numbers and types of compute tiles for multiple many-core configurations.

convolution are conducted with 64-bit integer data. Similar to 32-bit tile-based configurations, the computing scalability is approximately 2x by increasing the number of 64-bit compute tiles with the same factors.

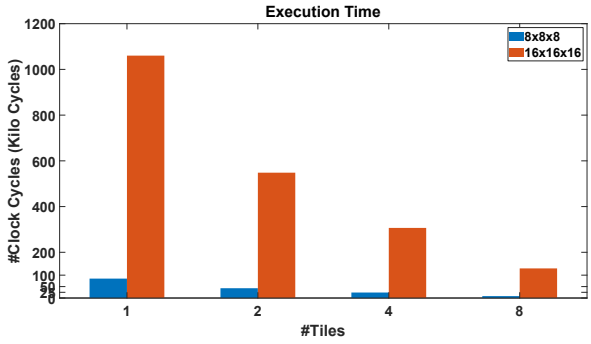
3-D convolution kernels

3-D parallel convolution kernels are considered for evaluation as they are commonly used for signal processing and neural network algorithms. 3-D parallel convolution kernels are evaluated over the aforementioned three many-core configurations based on 32-bit and 64-bit compute tiles. Also, 64-bit and 32-bit integer operations are supported by 32-bit and 64-bit compute tiles respectively. Parallel execution is conducted by partitioning the input matrix over the target number of compute tiles. In this use case, input matrices have square dimensions to be partitioned equally over a binary number of compute tiles. Inside each compute tile the input matrix is partitioned again over the number of PEs per tile. 3-D convolution is computed by a sliding window size of (3x3x3) across the assigned input matrix for each PE. Loading and storing operations from/to the memory during convolution are conducted using shared data memory for each PE due to the limited size of local data memory for larger sizes of input matrices to be executed.

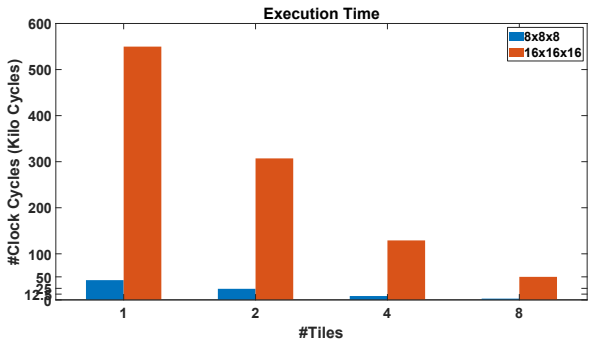
Figure 3.17 shows the execution time of several 3-D convolution kernels with different input matrix sizes (8x8x8, 16x16x16). Figure 3.17 (a) shows execution time using only the many-core configuration with 32-bit compute tiles. Therefore, the 3-D convolution parallelization is conducted over 4xRV32PEs up to 32xRV32PEs. The computing scalability is approximately 2x by increasing the number of compute tiles with the same factors. On the other hand, as



(a) 3-D convolution kernels, 32-bit tiles (4-PEs).



(b) 3-D convolution kernels, 64-bit tiles (1-PEs).



(c) 3-D convolution kernels, 64-bit tiles (2-PEs).

Figure 3.17: Execution time of several 3-D convolution kernels with different sizes over different numbers and types of compute tiles for multiple many-core configurations.

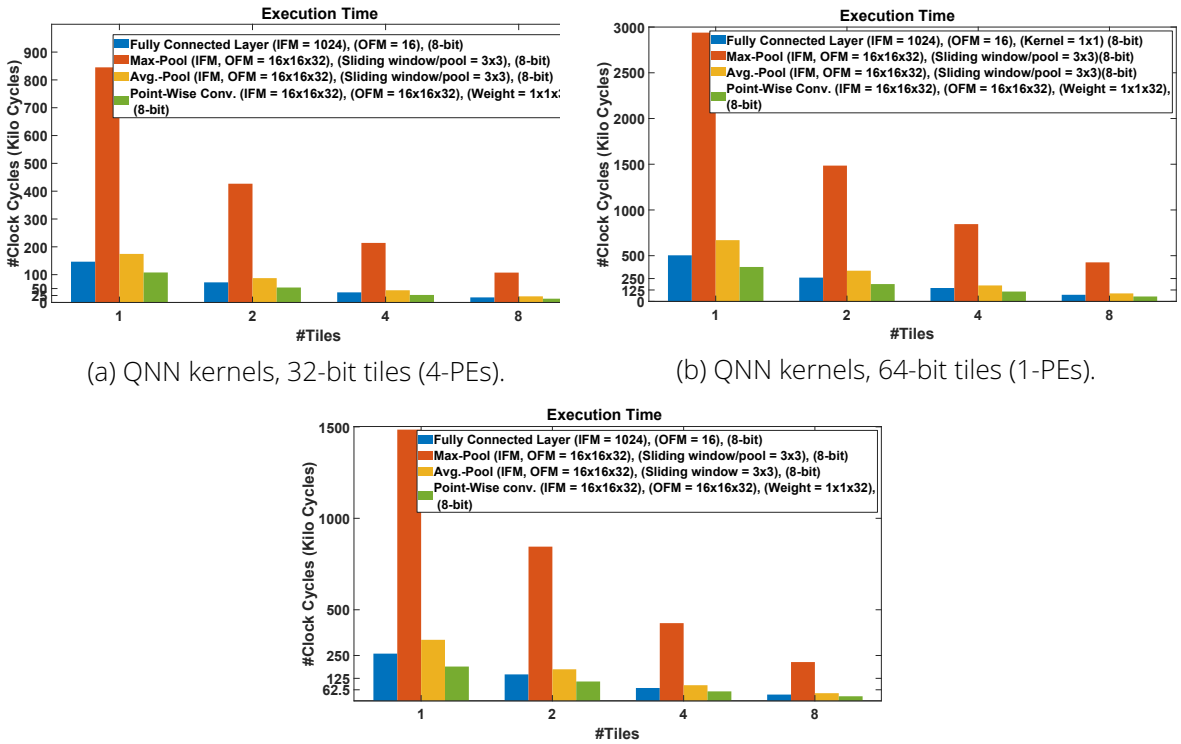
shown in Figure Figure 3.17 (b, c), the execution time over 64-bit compute tiles are higher in comparison with 32-bit compute tiles as the number of PEs per tile is less and the 3-D convolution are conducted with 64-bit integer data. Similar to 32-bit tile-based configurations, the computing scalability is approximately 2x by increasing the number of 64-bit compute tiles with the same factors.

Quantized neural network kernels

Four QNN kernels are derived from the open-source PULP-NN QNN library [172] for the proposed tile-based many-core evaluation. All QNN kernels are 8-bit quantized. The four used kernels are listed as follows:

1. Fully connected kernel: It is a simple matrix by vector multiplication, it generates a set of neurons as output (output feature map (OFM)). The input vector size (input feature map (IFM)) of the implemented fully connected layer is multiplied by a weight matrix to create the OFM. For the implemented fully connected layer: the IFM size = 1024, and the OFM size = 16. Unsigned 8-bit integer number for the IFM is fed as inputs along with the weights which are unsigned 8-bit integer of a size equal to (1024x16).

2. Max-pooling kernel: A pooling layer is a new layer preceding the convolutional layer which is applied to feature maps. Max-pooling calculates the maximum value for each patch on the received IFM. For the implemented max-pooling kernel: the IFM = 16x16x32, the OFM = 16x16x32, the sliding window/pool size = 3x3, and stride = 1 with zero padding.



(a) QNN kernels, 32-bit tiles (4-PEs).

(b) QNN kernels, 64-bit tiles (1-PEs).

(c) QNN kernels, 64-bit tiles (2-PEs).

Figure 3.18: Execution time of several QNN kernels with different sizes over different numbers and types of compute tiles for multiple many-core configurations.

3. Average-pooling kernel: The average-pooling kernel calculates the average value for each patch on the received IFM. For the implemented average-pooling kernel: the IFM = 16x16x32, the OFM = 16x16x32, the sliding window/pool size = 3x3, and stride = 1 with zero padding.

4. Point-wise convolution kernel: Point-wise convolution is 1-D convolution, which is parallelized according to height and width-wise over compute tile/PEs. For the implemented point-wise convolution, unsigned 8-bit integer of 16x16x32 are fed, where the spatial dimension is equal to 16x16 and there are 32 input channels. The IFM size = 16x16x32, the OFM size = 16x16x32, and the weight size = 1x1x32. The channel output is equal to 32, and the output spatial dimension is equal to 16x16.

For point-wise convolution and fully connected layer parallelization over the many-core compute tiles, the workload is partitioned based on the height dimension of the output feature map. The output height is divided by the number of available compute tiles as well as over the number of PEs per compute tile. The parallelization of the average pooling and the maximum pooling kernel is straightforward. The partitioning is based on multiple chunks to be assigned to each compute tile based on the size of the IFM. Both pooling functions do not have any weight parameter. They are just composed by a sliding window of 3x3 over the IFM which is an unsigned 8-bit integer for 16x16x32.

Figure 3.18 shows the execution time of QNN kernels with different parameters and IFM/OFM sizes. Figure 3.18 (a) shows execution time using only the many-core configuration with 32-bit compute tiles. Therefore, the QNN parallelization is conducted over 4xRV32PEs up

to 32xRV32PEs. The computing scalability is approximately 2x by increasing the number of compute tiles with the same factors. On the other hand, as shown in Figure Figure 3.18 (b, c), the execution time over 64-bit compute tiles is higher in comparison with 32-bit compute tiles as the number of PEs per tile is less. Similar to 32-bit tile-based configurations, the computing scalability is approximately 2x by increasing the number of 64-bit compute tiles with the same factors.

3.5 Summary

Chapter 3 presents a novel modular tile-based many-core architecture for heterogeneous ISAs by supporting multiple general-purpose RISC-V-based cores with different ISAs for FPGA devices. The proposed tile-based many-core architecture features a high degree of design scalability and regularity using heterogeneous RISC-V PEs for multi-/single-core compute tiles. The tile-based many-core architecture is based on a modular and configurable set of heterogeneous compute tiles connected through a scalable NoC architecture. Each compute tile supports scratchpad shared and local memory subsystems. Moreover, the proposed tile-based many-core architecture supports design-time configurations to change numbers and types of compute tiles for several many-core configurations. The proposed tile-based many-core architecture aims to ease the development and realization of multi-ISAs general-purpose many-core architectures by reducing the design time and the non-recurrent engineering costs. The tile-based architecture is evaluated based on hardware resource utilization, achievable memory bandwidth, and computing performance scalability through several many-core configurations and benchmarks. The results show a high degree of computing scalability using a scalable number of heterogeneous compute tiles.

Section 3.1 presents the architecture of the tile-based many-core platform. The many-core architecture consists of two hierarchical levels. The first level is the top-level or the NoC-based level, where multiple compute tiles are connected through a 2-D mesh NoC architecture. The second level of the hierarchy is the intra-tile architecture. Where PEs and scratchpad memories are hosted. There are three types of heterogeneous compute tiles supported by the proposed tile-based architecture. The first type is a 32-bit compute tile. Where the 32-bit compute tile internal architecture is a 32-bit multi-core architecture with scratchpad shared data/instruction memories connected through an AXI-4 interconnect. The 32-bit compute tile supports quad RV32 PEs. Each PE includes a single RV32IMC core with tightly coupled instruction and data memories to increase data locality and computing performance.

The second type is a 64-bit compute tile. Where the 64-bit compute tile internal architecture can be configured during design time to support a 64-bit single-core or a 64-bit dual-core architectures with scratchpad shared data/instruction memories connected through an AXI-4 interconnect. The 64-bit compute tile supports single or dual RV64 PEs. Each PE includes a single RV64IMAC core with tightly coupled instruction and data memories to increase data locality and computing performance. The third type is the main processing tile that supports a quad-core RV64 architecture with shared instruction memory and shared data memory connected through an AXI-4 interconnect. The shared data memory is an external DDR peripheral connected to the tile through a DDR controller. The main processing tile is the permanent compute tile in the tile-based architecture with a fixed number of RV64 PEs. The main processing tiles support the interfacing with several external peripherals such as UART and SD-card.

Section 3.2 presents the system scalability and the developed communication model over the NoC for data transfer and communication between different compute tiles. A lightweight circuit-switching NoC is used with low area overhead and high data rate. The NoC is based on a 2-D mesh architecture where the number of routers, NoC mesh size, and data packet size are configurable during design time. A unified NI is developed and implemented to provide TX/RX connection channels between compute tiles and NoC routers. The NI supports simultaneous data transmission and receiving between the NoC and compute tiles with two NI channels for TX and RX. A communication model is developed based on a message-based approach where data are transferred through data messages included within NoC data packets. The communication model is considered as the network transport layer over the NoC hardware architecture to control the data flow between compute tiles and maintain proper data transmission.

Section 3.3 presents the developed bare-metal parallel programming method for the proposed tile-based many-core architecture to generate multiple binary files from multi-tasks applications to be executed on many-core compute tiles. Each compute tile executes a separate binary file for its mapped task. We consider static task mapping over the selected number and type of compute tile that is conducted by the programmer/user prior to applications execution. Each compute tile is programmed individually from other compute tiles and the shared instruction memory is used as a boot memory for each compute tile.

Finally, Section 3.4 presents the evaluation and obtained experimental results of the proposed tile-based many-core architecture. The tile-based many-core architecture is evaluated based on hardware resource utilization, memory bandwidth and computing performance scalability using different signal processing kernels running over multiple numbers and types of compute tiles. The tile-based many-core architecture has been developed and implemented using a modular and hierarchical design approach. Architectural modules (i.e. RISC-V cores, PEs, memory blocks, interconnects, compute tiles, etc.) are implemented as IP components to be integrated together to build heterogeneous compute tile modules for several many-core configurations. Evaluation results demonstrate that the proposed tile-based many-core architecture features a scalable computing performance up to 685 MOPS for 8x32-bit compute tiles and 316 MOPS for 8x64-bit compute tiles with a scalable memory bandwidth up to 7.4 GB/s.

4 Towards Accelerator Memory Reuse Through a Hybrid Memory/Accelerator Tile Architecture

Today multi-/many-core SoC architectures can achieve high computing performance and energy efficiency due to their intrinsic heterogeneity. Specialized hardware accelerators coupled with big, and little general-purpose (GP) cores constitute the main compute tiles for modern many-core systems. Alongside, memory and input/output peripherals tiles provide data sharing and interaction between compute tiles and external peripherals. Loosely-coupled accelerator (LCA) model is increasingly used in heterogeneous architecture to achieve an order of magnitude high computing performance. However, LCAs require a large portion of private local scratchpad memory with the accelerator logic inside the custom accelerator tile architecture.

Accordingly, the increasing number of accelerator tiles leads to a significant increase in accelerators' private local memory (PLM) resources. For FPGA-based many-core systems, block memories (BRAMs, URAMs) are used to implement PLMs which have limited availability on FPGAs. Therefore, memory sharing between accelerator tiles and general-purpose tiles is necessary to reduce many-core systems' memory footprint. As a result, an increase in design complexity is associated with the growing number of heterogeneous tiles and their data patterns. Therefore, system-level design optimization and design modularity are key components for modern heterogeneous SoC architectures.

This Chapter presents a hybrid memory/accelerator tile architecture for FPGA-based many-core systems. The proposed hybrid tile is a modular and reusable tile that can be configured at run-time to operate as a scratchpad shared memory between many-core's compute tiles or as a LCA tile with one local custom hardware accelerator logic. In this chapter, a homogeneous FPGA-based RISC-V many-core platform with RV32 ISA [15] is used to integrate and evaluate the hybrid memory/accelerator tile architecture. The hybrid tile is implemented specifically for Xilinx FPGAs using Xilinx on-chip BRAM and URAM on-chip block memory.

The hybrid tile aims to support non-coherent memory sharing between compute tiles by partially reusing the LCA's PLM during its inactive time. Consequently, the hybrid tile supports non-coherent memory sharing between LCA inside the tile and many-core compute tiles. The proposed hybrid tile is implemented and evaluated on a Xilinx Ultrascale+ FPGA. A set of signal processing based kernels and custom hardware accelerators are used to evaluate

the hybrid tile in terms of data transfer latency over the NoC to/from RISC-V-based compute tiles. The hybrid tile architecture features a low resource utilization overhead which reduces the cost of scalability for many-core systems realization.

The Chapter is structured as follows. Section 4.1 describes the hybrid memory/accelerator tile architecture including internal components of the tile data path and control unit. Also, seamless integration of stream-based hardware accelerator inside the tile data path architecture. Section 4.2 presents how the hybrid tile is integrated into tile-based many-core systems and describes controlling and data transfer techniques between compute tiles and the hybrid tile through the NoC. Section 4.3 presents the hybrid tile evaluation in terms of resource utilization, memory bandwidth and data transfer rate with several signal processing based kernels and custom stream-based hardware accelerators. Finally, Section 4.4 summarizes this chapter.

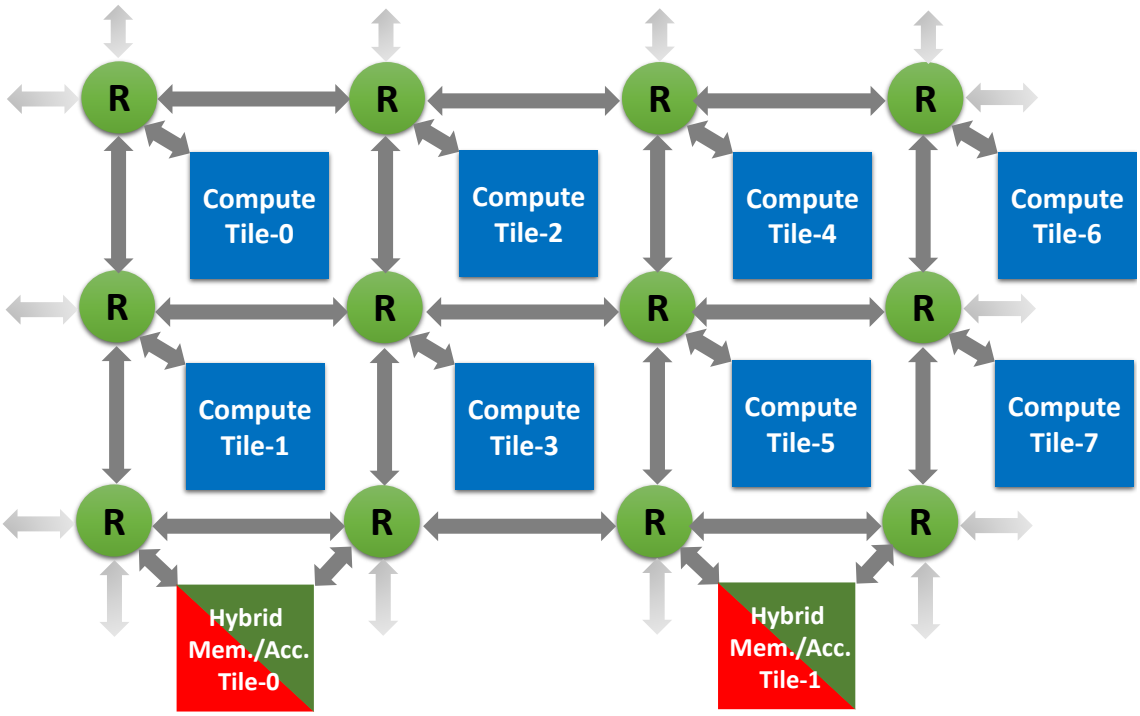


Figure 4.1: An overview of a heterogeneous tile-based many-core architecture with hybrid memory/accelerator tiles. The many-core system supports a single ISA by homogeneous RISC-V cores with heterogeneous LCAs hosted by hybrid memory/accelerator tiles.

4.1 Hybrid Tile Architecture Implementation

To support seamless integration of LCAs through NoC for tile-based many-core architecture, a modular and parametrized tile architecture must be developed and implemented to support the integration of LCAs with different resource and memory sizes. In addition, to efficiently use on-chip memory on the FPGA floorplan, accelerator memory reuse is a suitable technique to reuse the accelerator's PLM during inactive time as a shared memory between compute tiles

or a scratchpad storage extension for any compute tiles. Therefore, in this section, a hybrid memory/accelerator tile architecture is presented supporting two modes of operations. The first mode is the memory mode where the tile can be used as a shared scratchpad memory between compute tiles through the NoC. The second mode is the accelerator mode where the tile is acting as a LCA tile hosting a single stream-based custom hardware accelerator.

The hybrid memory/accelerator tile is designed and implemented to be seamlessly integrated into NoC-based many-core architectures where the NoC is the interconnection medium between all many-core tiles (including RISC-V based compute tiles and hybrid memory/accelerator tiles). In this chapter, the target tile-based many-core system consists of a scalable number of homogeneous RISC-V based compute tiles as shown in Figure 4.1. Each compute tile supports four RISC-V cores based on RV32IMC ISA.

The RISC-V based compute tiles are responsible to send application/user-defined control messages through the NoC to the hybrid tiles for configuration at run-time. Control messages contain tile and memory configuration parameters to configure the hybrid tile based on compute tile message request to access hybrid tile on-chip memory for read/write (R/W) or to activate the local accelerator logic. Similarly, read and write data from/to the hybrid tile are transferred in the form of data packets through the NoC. The hybrid memory/accelerator tile architecture is connected to the NoC via two separate network interfaces (NIs). The first NI is dedicated to sending and receiving control packets (NI-Ctrl) and the second NI is for data transfer (NI-Data) from/to many-core compute tiles.

The reason behind using two NIs is to avoid data and control messages overlapping during transmission, which increases data transfer bandwidth over the NoC and allows parallel and simultaneous transfer of control and data packets from/to multiple compute tiles at the same time. Each NI includes two separate channels for data/control packets transmission and receiving (NI-TX, NI-RX). NIs are implemented using parametrized size AXI-S FIFO IPs supporting 32-bit AXI-stream interface for control and data packets. The 32-bit AXI-stream

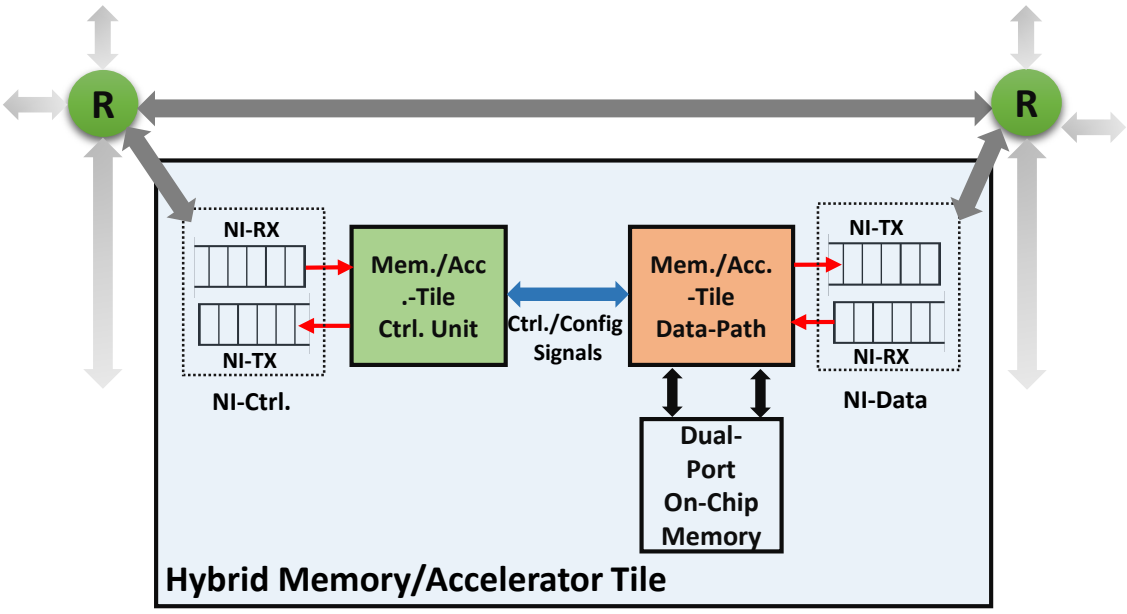


Figure 4.2: An overview of the hybrid memory/accelerator tile internal architecture showing control unit, data path, and data/control NIs to NoC routers.

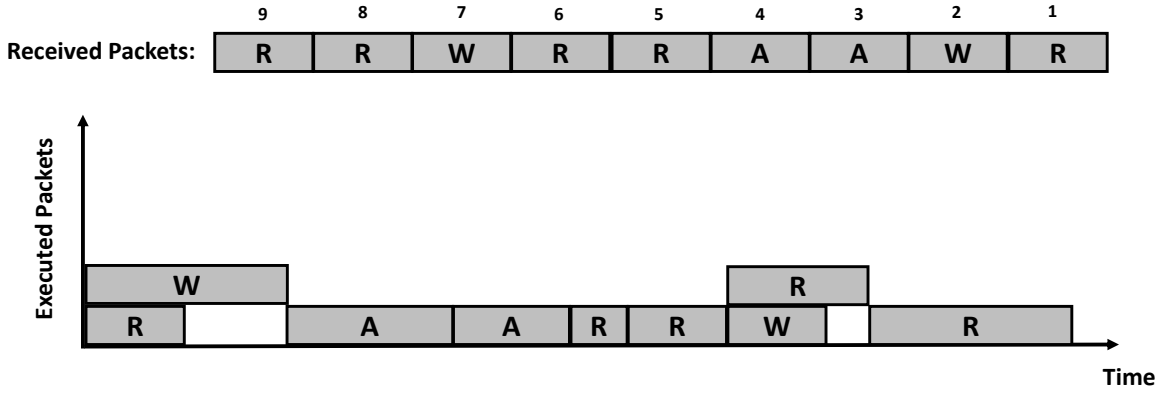


Figure 4.3: An example of a received sequence of message requests and their order of execution by the hybrid memory/accelerator tile.

interface is compatible with the NoC router interface. The hybrid memory tile consists of several internal architectural components as shown in Figure 4.2. The internal architecture components are listed as follows:

- Two separate NIs: a NI for read and write data packets, and a NI for control packets that sends and receives control messages through a lightweight NoC architecture.
- A control unit that decodes received control messages and sends configuration signals to the hybrid tile data path based on the requested mode of operations. Also, it sends control messages carrying grant signals to compute tiles indicating if the request message can be processed or not.
- A data path that includes the hybrid tile functional units: memory read and write managers, hardware accelerator wrapper, on-chip memory access blocks, and multiplexers/demultiplexers.
- Parametrized size of dual-ported on-chip block memory (BRAMs/URAMs) that are used either as a PLM for accelerator or shared scratchpad memory between compute tiles.

The hybrid memory/accelerator tile is capable to process two simultaneous read and write memory requests at the same time due to the utilization of dual-ported on-chip memory. In the case of an accelerator mode request, a single accelerator message request can be served solely to allow the hosted accelerator logic to fully utilize the on-chip memory as a PLM for memory read and write operations without any overlapping with other memory requests. An example of a received sequence of request messages by the hybrid tile NI-Ctrl buffer is shown in Figure 4.3. In this example, a received sequence of various request message types is stored first in the NI-Ctrl buffer before the decoding stage to identify and categorize the received messages based on the mode of operation and define their process order. The

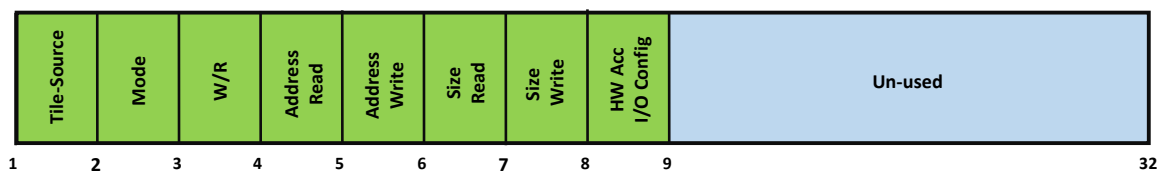


Figure 4.4: Structure of hybrid memory/accelerator tile request message.

request message structure is shown in Figure 4.4. It consists of a data array of 32 locations where each location stores 32-bit data. The reason for using a message length of 32 is to be compatible with the used NoC packet size where a single NoC packet contains 32 flits of data. The first eight locations are used to store the hybrid tile configuration parameters and the tile source address, the rest of locations are unused and have to be filled with zeros to create the required NoC packet length for packet transferring over the NoC. There are three types of message requests based on the mode of operation: 1) Memory read request, 2) memory write request, and 3) accelerator mode request. Accordingly, and as shown in Figure 4.3, two consecutive memory read and write requests can be processed simultaneously. But in case of two consecutive message requests from the same type, the two message requests are processed sequentially based on the time of arrival such as two consecutive memory read or write requests. On the other hand, accelerator message requests are processed individually to provide full access to on-chip memory by the accelerator logic. Also, consecutive memory and accelerator message requests are processed sequentially based on the time of arrival.

4.1.1 Hybrid Tile Data Path

The tile data path is responsible to establish data paths between NI-Data (RX/TX) and on-chip memory based on received message requests. The control unit decodes message requests and generates the corresponding control signals to setup and configure the data path based on the mode of operation. Moreover, the tile data path includes on-chip memory access blocks that receive memory configuration parameters (i.e. addresses, data sizes) for reading and writing data from the on-chip BRAM/URAM blocks. The tile data path consists of several functional and data movement components to establish the necessary data path between the NI-Data and on-chip memory. Figure 4.5 shows a detailed block diagram of the tile data path and its internal components. The tile data path supports three types of data paths based on the decoded message process as shown in Figure 4.5. The three data paths types are listed as follows:

- Memory write (*mem_W_mode*) mode data path: A memory write data path is established based on a *mem_W_mode* request to connect the NI-RX to the dual ported on-chip memory. It provides access to hybrid tile on-chip memory from any compute tile in the many-core system through the NoC to write a specific size of data from a specific memory address based on the transmitted memory configuration parameters. Memory write data path supports 32-bit stream data transfer to on-chip memory through a memory write manager and on-chip memory access units as shown in Figure 4.5.
- Memory read (*mem_R_mode*) mode data path: A memory read data path is established based on a *mem_R_mode* request to connect the dual ported on-chip memory to the NI-TX. It allows access to hybrid tile on-chip memory from any compute tile in the many-core system through the NoC to read a specific size of data from a specific memory address based on the transmitted memory configuration parameters. Memory write data path supports 32-bit stream data transfer to on-chip memory through a memory write manager and on-chip memory access units as shown in Figure 4.5.
- Accelerator (*A_mode*) mode data path: A LCA can be activated through the accelerator data path based on an *A_mode* request to connect the accelerator wrapper I/O to the NI-RX and NI-TX as well as the accelerator wrapper memory ports to the on-chip memory. In this mode of operation, any compute tile in the many-core system can

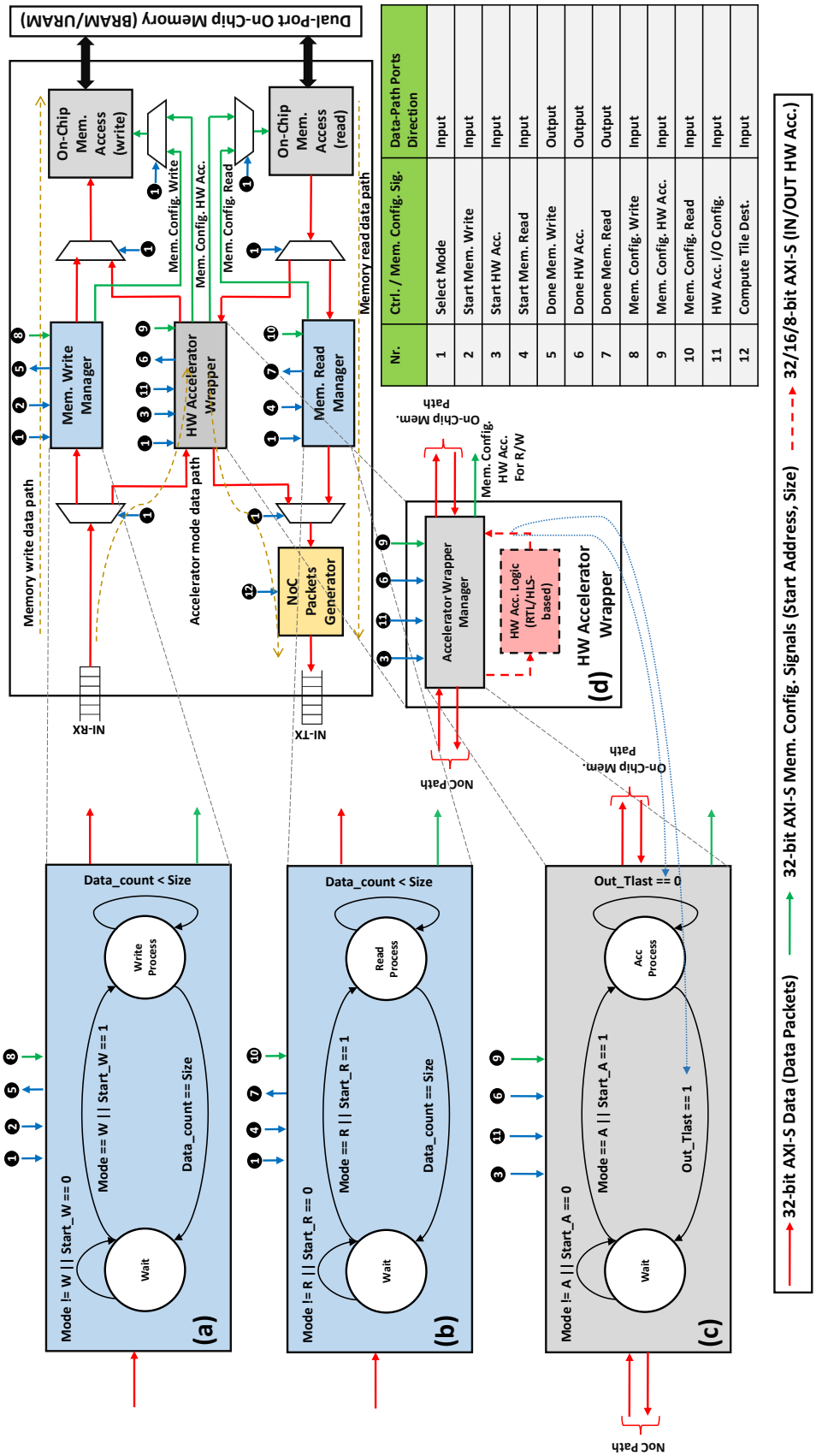


Figure 4.5: A detailed block diagram of hybrid memory/accelerator tile data path architecture showing internal functional and data movement components.

directly send an receive stream to/from the accelerator logic. Also, the accelerator logic can fully access the on-chip memory as a PLM for memory read and write operations. The accelerator mode data path supports 32-bit stream data transfer between compute tiles and accelerator logic. Also, on-chip memory data width and memory address size are 32-bit as shown in Figure 4.5.

Memory read and write paths can be established in parallel to handle read and write memory requests simultaneously to/from dual-ported on-chip memory (32-bit data, 32-bit address). On the other hand, only the accelerator path can be solely established as the hardware accelerator logic requires to use the on-chip memory as a PLM for load/store operations. The tile data path consists of several functional and data movement blocks in order to implement the different data path modes. A detailed description of tile data path internal components are described as follows:

Memory write manager

The memory write manager is responsible to manage the data writing process to the on-chip memory. As shown in Figure 4.5, the memory write manager block has three stream interfaces and four control I/O ports. The stream interfaces are for 1) 32-bit input data stream from the NI-RX, 2) 32-bit output data stream to the on-chip memory access, and 3) output stream interface carrying memory configuration parameters to the on-chip memory access. The four control signal ports are connected to the control unit block as shown in the right table in Figure 4.5.

The four ports are: 1) hybrid tile mode selection to select the memory write data path, 2) start signal (*start_w*) to start the data transfer process from the NI-RX to the on-chip memory access, 3) a done signal (*done_mem_w*) as an output to the control unit to indicate the successful transmission of the complete data size to the on-chip memory, and 4) memory configuration parameters extracted from the request message. The memory write manager implements a FSM that manages the data transfer process as shown in Figure 4.5 (a).

The FSM has two states, a wait state and a write process state. The wait state checks for the select mode and start signal from the control unit. In case the *select_mode == mem_w_mode && start_w == 1*, the next state is the write process state where the memory write manager starts to write the data to the on-chip memory. The *done_mem_w* signal is equal to one when the data transfer count (*data_count*) is equal to the requested data size to be written in the on-chip memory. Afterwards, the memory write manager moves to the wait state in order to wait for a new memory write request.

Memory read manager

The memory read manager is responsible to manage the data reading process from the on-chip memory. As shown in Figure 4.5, the memory read manager block has three stream interfaces and four control I/O ports. The stream interfaces are for 1) a 32-bit output data stream to the NI-TX, 2) a 32-bit input data stream from the on-chip memory access, and 3) an output stream interface carrying memory configuration parameters to the on-chip memory access.

The four control signal ports are connected to the control unit block as shown in the right table in Figure 4.5. The four ports are: 1) hybrid tile mode selection to select the memory read data path, 2) start signal (*start_r)*to start the data transfer process from the on-chip

memory access to the NI-RX, 3) a done signal (*done_mem_r*) as an output to the control unit to indicates the successful transmission of the complete data size from the on-chip memory, and 4) memory configuration parameters extracted from the request message.

The memory read manager implements a FSM that manages the data transfer process as shown in Figure 4.5 (b). The FSM has two states, a wait state and a read process state. The wait state checks for the select mode and start signal from the control unit. In case the *select_mode == mem_r_mode && start_r == 1*, the next state is the write process state where the memory read manager starts to read the data from the on-chip memory. The *done_mem_r* signal is equal to one when the data transfer count (*data_count*) is equal to the requested data size to be read from the on-chip memory. Afterwards, the memory read manager moves to the wait state in order to wait for a new memory read request.

Hardware accelerator wrapper

The accelerator wrapper is responsible to host and manage the hardware accelerator logic. It supports seamless integration of RTL-/HLS-based accelerator with I/O stream interfaces as shown in Figure 4.5 (d). The accelerator wrapper block has five stream interfaces and five control I/O ports. Control signals are responsible to control stream data flow to/from the accelerator logic and accelerator wrapper interfaces to on-chip memory and NI-Data. Four input/output (I/O) configurations are supported based on the request packet control for accelerator mode: 1) NoC-mem, 2) NoC-NoC, 3) mem-NoC, and 4) mem-mem. Therefore, data can be streamed directly between local accelerator logic and a single compute tile by setting I/O interfaces to NI-Data.

The stream interfaces are for 1) 32-bit input data stream reading from the NI-RX, 2) 32-bit input data stream writing to the NI-RX, 3) 32-bit output data stream to the on-chip memory access (PLM storing operations), 4) 32-bit input data stream from the on-chip memory access (PLM loading operations), and 5) output stream interface carrying memory configuration parameters to the on-chip memory access for PLM load/store operations.

The five control signal ports are connected to the control unit block as shown in the right table in Figure 4.5. The five ports are: 1) hybrid tile mode selection to select the accelerator mode data path, 2) start signal (*start_a)* to activate the hosted accelerator logic, 3) a done signal (*done_a*) as an output to the control unit to indicate the successful completion of the accelerator function, 4) memory configuration parameters extracted from the request message, and 5) hardware accelerator I/O configuration to select I/O direction to the NI-Data to allow the compute tiles to send and receive data directly to the hardware accelerator logic.

The accelerator wrapper consists of an accelerator wrapper manager, and the hardware accelerator logic. The accelerator wrapper manager is responsible to control stream data flow to/from the accelerator logic and accelerator wrapper interfaces to on-chip memory and NI-Data.

The accelerator wrapper manager implements a FSM that manages the data transfer process and the activation of accelerator logic as shown in Figure 4.5 (c). The FSM has two states, a wait state and an accelerator process state. The wait state checks for the select mode and start signal from the control unit. In case the *select_mode == a_mode && start_a == 1*, the next state is the acceleration process state where the accelerator starts to access the on-chip memory based on load/store operations from the hosted logic functions. The *done_a* signal

is equal to one when the acceleration function has been done by checking the accelerator output stream interface (*out_tlast == 1*). Afterwards, the accelerator wrapper manager moves to the wait state in order to wait for a new accelerator mode request.

NoC packet generator

The tile data path has a NoC packet generator to create data packets in order to transmit the read data over the NoC to compute tiles destinations. The packet generator receives a data stream either from the accelerator wrapper or the memory read manager and split them into data flits. Each data packet consists of 32 data flits with a header flit that contains the compute tile destination. The packet generator streams the data to the NI-TX where each data packet requires four clock cycles to be created.

On-chip memory access units

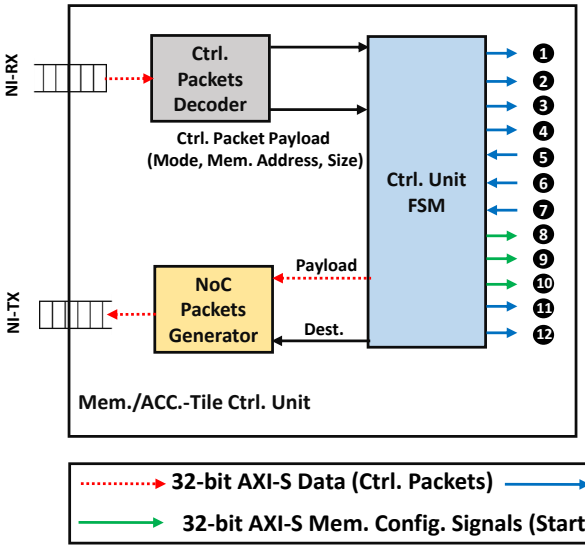
There are two on-chip memory access units inside the tile data path to convert the stream interfaces to be compatible with on-chip BRAM/URAM interfaces. The first on-chip memory access unit is for writing data to on-chip memory. It receives a stream of data with memory configuration parameters in order to write the data on the corresponding memory address. The second on-chip memory access unit is for reading data from on-chip memory. It sends a stream of data from the on-chip memory either to the accelerator wrapper or to the memory read manager. The memory access block converts the block memory interfaces to a stream interface using memory configuration parameters to read a specific size data from a specific memory address.

4.1.2 Hybrid Tile Control Unit

The control unit handles all memory and accelerator message requests from all compute tiles in the tile-based many-core system as shown in Figure 4.2. It sends and receives control, response, and configuration signals to the tile data path unit to ensure proper data transfer between the hybrid tile and compute tiles destination. Also, it is considered the tile data path manager. The control unit is responsible for setup, configuring, and monitoring the tile data path during operations. The three types of data paths that are mentioned before are established based on the control and configuration signals generated from the control units based on received message requests. The control unit is capable to decode two received message requests simultaneously and generate their corresponding control and configuration signals to the tile data path based on request modes of operation. The control packet payload contains the request message which includes all necessary information to configure the tile data path based on the requested mode of operation, and memory configurations for read and write data from/to memory blocks. The control unit internal architecture is shown in Figure 4.6. It consists of three main architectural blocks: 1) the control packets decoder, 2) the control unit FSM, and 3) the NoC packet generator. A list of all control and configuration signals is shown in the right table of Figure 4.6.

The control packets decoder

The decoder unit is the first stage that receives control packets from the NI-Ctrl-RX sent from compute tiles carrying request messages. The received control packets are fetched based



Nr.	Ctrl. / Mem. Config. Sig.	Ctrl. Unit FSM Port
1	Select Mode	Output
2	Start Mem. Write	Output
3	Start HW Acc.	Output
4	Start Mem. Read	Output
5	Done Mem. Write	Input
6	Done HW Acc.	Input
7	Done Mem. Read	Input
8	Mem. Config. Write	Output
9	Mem. Config. HW Acc.	Output
10	Mem. Config. Read	Output
11	HW Acc. I/O Config.	Output
12	Compute Tile Dest.	Output

Figure 4.6: A detailed block diagram of hybrid memory/accelerator tile control unit architecture.

Table 4.1: Input and output of the decoding stage in the control unit.

Decoding Unit Input				Decoding Unit Output			
Mode Msg_1	Mode Msg_2	R/W Msg_1	R/W Msg_2	Messages Category	Msg_1 Type	Msg_2 Type	Message Count
Mem.	Mem.	R/W	W/R	RW/WR	Read/Write	Write/Read	2
Mem./Acc.	Acc./Mem.	R/-	-/R	RA/AR	Read/Acc.	Acc./Read	2
Mem./Acc.	Acc./Mem.	W/-	-/W	WA/AW	Write/Acc.	Acc./Write	2
Acc.	Acc.	-	-	AA	Acc.	Acc.	2
Mem.	Mem.	R	R	RR	Read	Read	2
Mem.	Mem.	W	W	WW	Write	Write	2
Acc.	-	-	-	A	Acc.	-	1
Mem.	-	R	-	R	Read	-	1
Mem.	-	W	-	W	Write	-	1

on a fixed priority upon arrival to the packets decoder unit to extract the payload of the received control packet. The decoder unit can fetch two control packets at a time to handle read and write requests simultaneously by the tile data path. The decoding stage starts by extracting modes of operation from the two received request messages to categorize them based on request modes of operations. Nine message categories (*msg_cat*) are supported by the decoding stage, (*msg_cat: (RW/WR, RA/AR, WA/AW, AA, RR, WW, A, R, W)*). Table 4.1 shows

the decoding stage input and output data. The input data are extracted from the received request messages and the output data is forwarded to the control unit FSM to generate the control signals accordingly. Also, the decoding stage provides the type of each request message as mentioned in Table 4.1 (column 6, 7) as well as the message count. The message type (*msg_type*) defines the mode of operation of each message as memory read, write, or accelerator requests. The message count by default is equal to two as the tile can process two request messages simultaneously. But in case only one request message is received by the NI-RX, the message count is equal to one.

The control unit FSM

The control unit operates based on a FSM that handles reading packets, decoding packets, and processing packet requests to generate the required control and configuration signals to the tile data path. Figure 4.7 shows the control unit main FSM. The main FSM consists of four states described as follows:

- Read packets state: The read packets state is the starting state that read either two packets or one packet based on packet availability in the (*NI_buffer*). The read state extracts the request messages from the received packet and stored them in an internal buffer. The reading state checks for *packet_count* to be equal to two or the *NI_buffer* is empty to move to the next state which is the decode state for received request messages decoding.
- Decode messages state: The decode messages state is responsible to manage the decoding stage to generate messages categories as shown in Table 4.1. The decoding state also checks for the tile data path done signals (*done_r, done_w, done_a*) to be one in order to move to the process message requests. In case the tile data path is busy processing previous message requests, the next state is a waiting state until the tile data path is free to process the current message requests.
- Wait state: The wait state checks the done signals from the tile data path (*done_r, done_w, done_a*) to be one in order to move to the process messages request state.
- Process message requests state: The process state is responsible to control and configure the tile data path based on message categories received from the decoding state. Also, this state monitors the data path operating states (i.e. busy or free) in order to move to the read packets state in case all current message requests are fully processed. Figure 4.8 shows the internal FSM of the process state. A description of the messages process FSM is presented as follows:
  - Read message category state: In this state messages category are received from the decoding state. Afterward, the control unit sends a response message to the compute tile source (request message source) with a grant message informing the compute tile that a memory/accelerator data path is established for memory reading/writing or accelerator mode. The grant message is a 32 location message, a similar size to the response message carrying grant signals to the compute tile which sent the request message. Based on the received messages category as shown in Figure 4.8 the next state is selected.
  - Read and write state: In case (*msg_cat == (RW)||(WR)*), the read and write state sends memory select mode (*sel_mode == mem*), and start signals (*start_r* and *start_w*) to the tile data path memory read and write managers to establish memory read

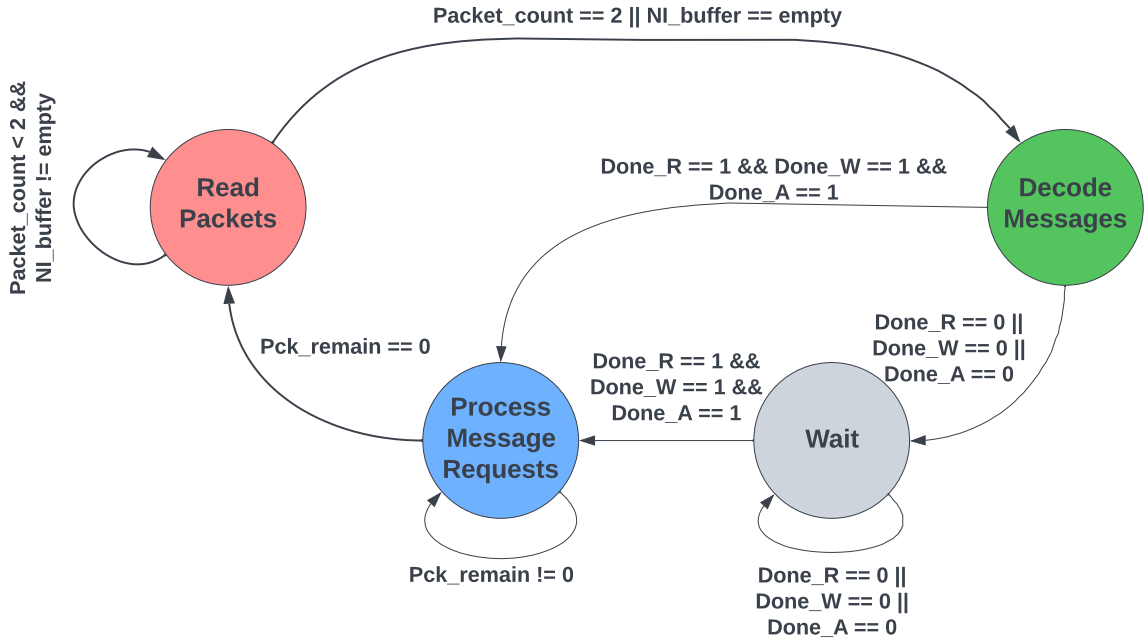


Figure 4.7: The main FSM of the hybrid tile shows the four stages of the control unit.

and write paths simultaneously. Also, memory configuration parameters are sent to the on-chip memory access units. After the read and write processes are conducted and no remaining messages need to be processed, the next state is the exit state.

- Read state: In case (*msg_cat* == *(RA)* | |*(RR)* | |*(R)*), the read state sends memory select mode (*sel_mode* == *mem*), and start signal (*start_r*) to the tile data path memory read manager to establish memory read data path. Also, memory configuration parameters are sent to the on-chip memory access read unit. According to the second message type (*msg_2_type*), the next state is determined. For example, if (*msg_2_type* == *R*) the next state is a read state. If (*msg_2_type* == *A*) the next state is an accelerator state. After the read process is conducted and no remaining messages need to be processed, the next state is the exit state.

- Write state: In case (*msg_cat* == *(WA)* | |*(WW)* | |*(W)*), the write state sends memory select mode (*sel_mode* == *mem*), and start signal (*start_w*) to the tile data path memory write manager to establish memory write data path. Also, memory configuration parameters are sent to the on-chip memory access write unit. According to the second message type (*msg_2_type*), the next state is determined. For example, if (*msg_2_type* == *W*) the next state is a write state. If (*msg_2_type* == *A*) the next state is an accelerator state. After the write process is conducted and no remaining messages need to be processed, the next state is the exit state.

- Accelerator state: In case (*msg_cat* == *(AW)* | |*(AR)* | |*(AA)* | |*(A)*), the accelerator state sends accelerator select mode (*sel_mode* == *acc*), and start signal (*start_a*) to the tile data path accelerator wrapper manager to establish accelerator data path and activate the accelerator logic. Also, memory configuration parameters and accelerator I/O configuration are sent to the accelerator manager and on-chip memory access

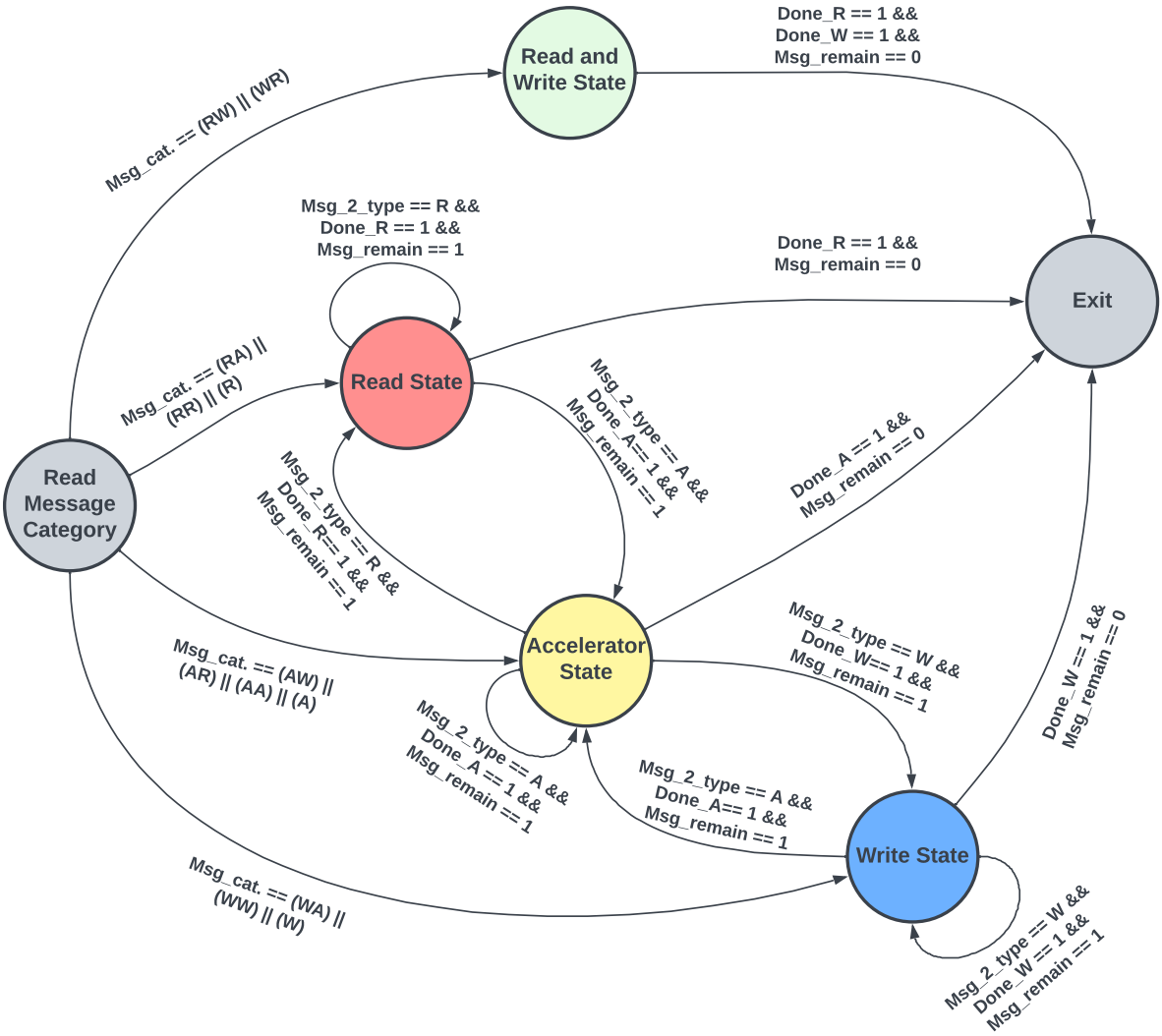


Figure 4.8: A detailed FSM of the messages processing stage.

read/write units. According to the second message type (*msg_2_type*), the next state is determined. For example, if (*msg_2_type* == *W*) the next state is a write state. If (*msg_2_type* == *A*) the next state is an accelerator state. After the accelerator process is conducted and no remaining messages need to be processed, the next state is the exit state.

- Exit state: the exit state checks that all done signals are equal to one and no remaining messages need to be processed to move to the read packets state in order to process new message requests.

The NoC packet generator

the NoC packets generator is implemented, similar to the same unit in the tile data path, to create response packets to be transmitted over the NoC to the destination tiles. The packet generator receives response messages from the FSM unit and split them into data flits. Each data packet consists of 32 data flits with a header flit that contains the compute

tile destination. The packet generator streams the data to the NI-TX where each data packet requires four clock cycles to be created.

4.2 Integration into Tile-based Many-Core System

The hybrid tile is integrated into a homogeneous RISC-V-based many-core system [15]. The system is based on a NoC-based many-core architecture for FPGA devices as described in Chapter 3. In this section, the integration between the homogeneous RISC-V based many-core system and the hybrid memory/accelerator tile is described. Also, the message-based communication protocol for data and control messages transmission over the NoC between compute (RISC-V-based tiles) and hybrid tiles is presented as well as related software modules.

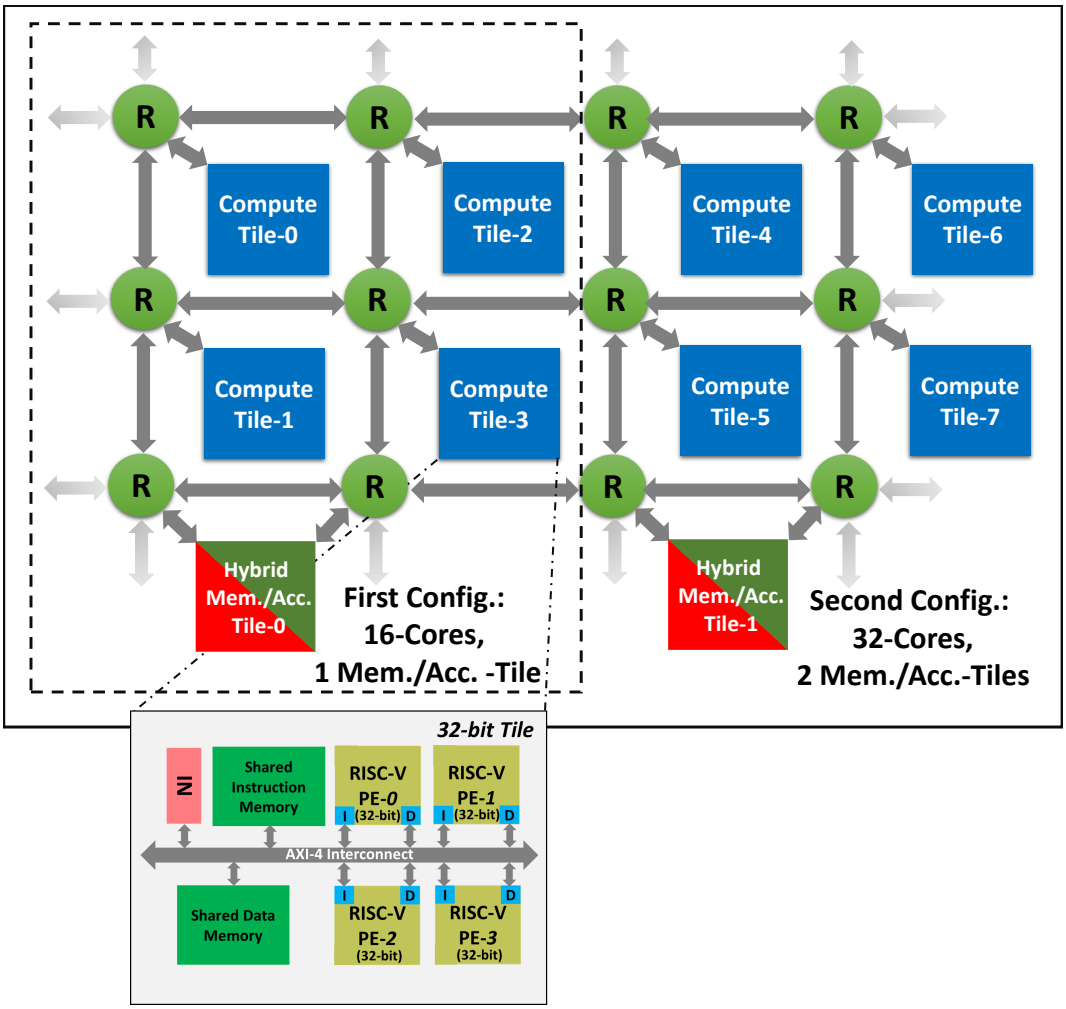


Figure 4.9: RISC-V based many-core configurations, configuration one: 16xRISC-V cores, and single hybrid memory/accelerator tile, configuration two: 32xRISC-V cores, and 2xhybrid memory/accelerator tiles

4.2.1 System Overview

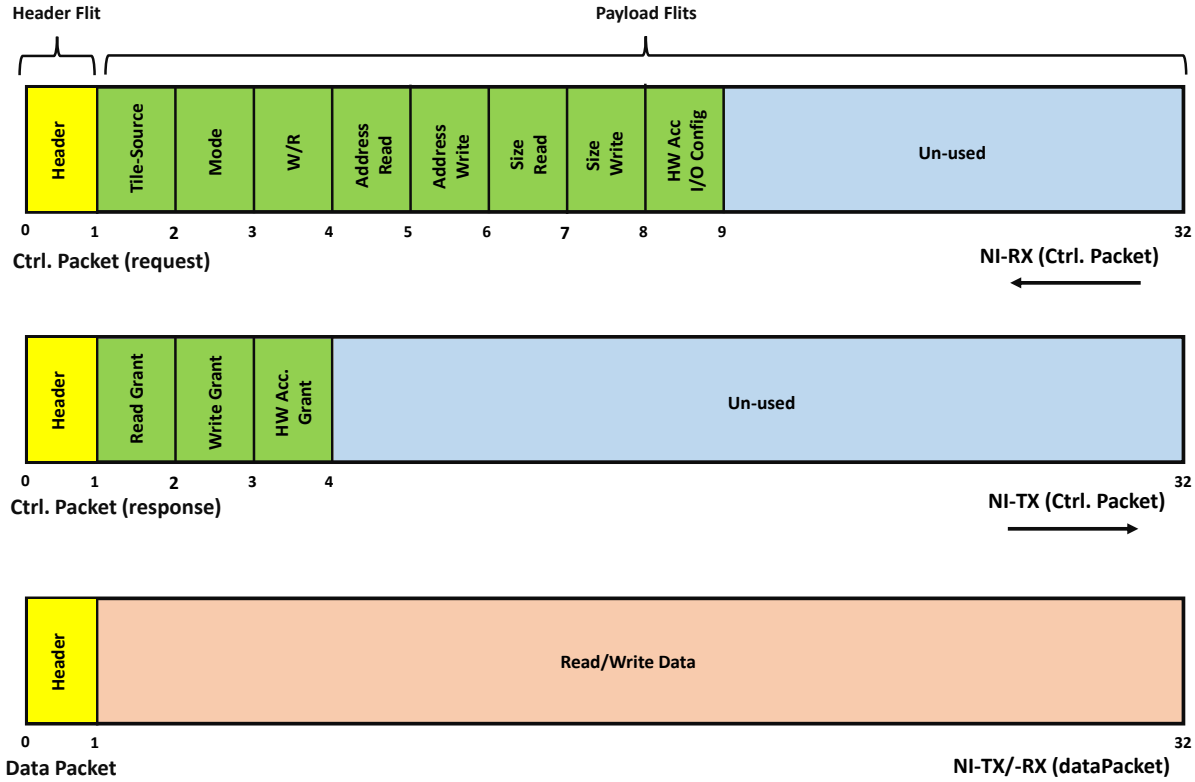
The homogeneous RISC-V-based many-core system consists of a scalable number of compute tiles, each compute tile is based on a soft quad-core RISC-V based on RV32IMC ISA (RI5CY core) [77]. In addition, compute tiles contain non-coherent shared data and instruction memories between RISC-V cores connected through an AXI-4 interconnect. Compute tiles are connected to the NoC through a single NI. The NI is connected to the AXI-4 interconnect as a memory-mapped peripheral to RISC-V cores. An overview of the target manycore system with hybrid memory tiles is shown in Figure 4.9. The hybrid memory/accelerator tile is connected to the NoC through two routers for data and control packets. Each router is connected to a NI with 32-bit AXI-S interface, the NI has two channels for TX and RX. The NI-RX contains a large FIFO of 8K locations to store the incoming packets prior to read packet process for both data and control packets. The whole system including RISC-V based compute tiles, the NoC, and hybrid tiles are running on the same clock frequency.

Two tile-based many-core configurations are realized based on the number of compute tiles/RISC-V cores and the number of hybrid memory/accelerator tiles. The first configuration consists of four compute tiles with 16 RISC-V cores and one hybrid memory/accelerator tile. The second configuration contains eight compute tiles with 32 RISC-V cores and two hybrid memory/accelerator tiles. All control messages are created and sent by software kernels running on RISC-V cores based on application requirements. During data transfer between a compute tile and the hybrid tile, the received or transmitted data is handled by compute tile NI which read or write the corresponding size of data from/to compute tile shared data memory. All compute tiles NIs are managed and controlled by software kernels for memory access operations inside the tile. A bare-metal programming method is supported by the tile-based many-core system for software kernels execution. For parallel kernels execution, static application mapping and partitioning are conducted prior to execution over RISC-V-based compute tiles.

4.2.2 Message-based communication over NoC

The interaction between the compute tiles and the hybrid memory/accelerator tile is conducted through a message-based communication protocol. Where the compute tiles request to access the on-chip memory or the accelerator through a set of control messages. Also, data transmission during memory read and write operations are conducted through data messages between tiles. Messages are created by a RISC-V core inside compute tiles and transmitted through the NoC to the destination tiles. For the tile-based many-core architecture, a circuit-switched-based NoC is used due to low-latency data transfer compared to a packet-switching NoC [11]. Data and control messages are transferred through the NoC in a form of packets. All packets consist of 33 flits, where each flit is 32-bit. The first flit is the header flit which contains the destination address *X_Y* for either NI-Ctrl or NI-Data, as the hybrid tile is connected to different NoC routers for control and data packets. The other 32 flits are the packet payload that contains data or control messages.

Three types of packets are supported as shown in Figure 4.10. First, the control request packet contains 8 flits carrying information for compute tile source address, requested mode of operation, memory read/read address, data write/read size, and hardware accelerator I/O



Config.	Mode	W/R	HW Acc. I/o Config
Read	0	0	-
Write	0	1	-
HW Acc.	1	-	0 >> (In, Out: NoC) 1 >> (In: NoC, Out: Mem.) 2 >> (In: Mem., Out: NoC) 3 >> (In, Out: Mem)

Figure 4.10: Structure of (1) request, (2) response control packets, and (3) data packets used by the hybrid memory/accelerator tile.

configurations. Second, the control response packet is sent by the hybrid memory/accelerator tile to compute tile source carrying grant signals for read, write, or accelerator ready. Lastly, data packets have a fixed structure of 32 flits of data payload for both read and write directions from/to hybrid memory/accelerator tiles. Based on mode of operations different software modules are used to control data transmission between compute tiles and hybrid tiles. Also, two different sequence diagrams for message-based communication protocol are supported for read and write data between compute tiles and hybrid tiles.

Memory read/write mode

For memory read/write mode a memory request message is created by a RISC-V core from the compute tile that request to access the on-chip memory on the hybrid tile. In order to transmit memory request messages, a communication protocol is developed to ensure proper messages transmission between a compute tile and hybrid tile. As shown in Figure 4.11, a sequence diagram of a memory read operation is presented. Where the compute tile sends a request packet through the NoC carrying the request message to the hybrid tile (NI-Ctrl). After a period of time, based on the hybrid tile data path status, a response packet

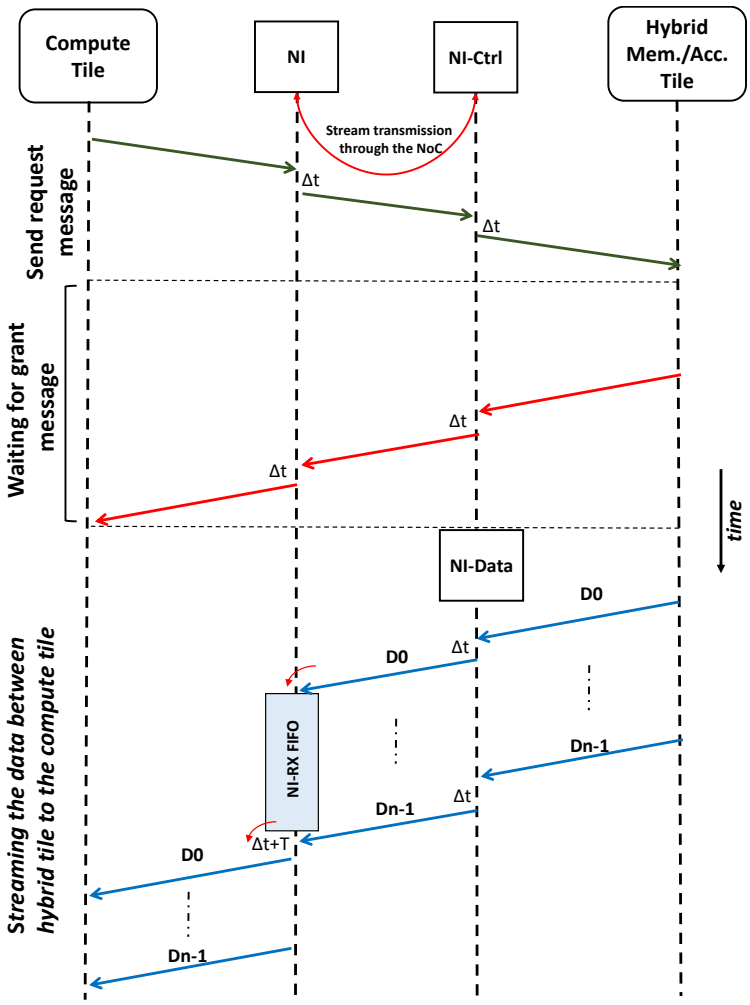


Figure 4.11: Sequence diagram of the data transfer process between a compute tile and hybrid memory tile in case of memory or accelerator data read.

is submitted to the source compute tile carrying the grant message. The grant message informs the compute tile about the readiness of the hybrid tile to respond to the sent request message. In case the grant message informs the compute tile that the hybrid tile can not respond to the submitted request, the compute tile resends again the request message till it receives a grant message to start receiving read data memory from the hybrid tile.

Listing 4.1 shows a detailed description of the request message software module executed by a RISC-V core from compute tiles. Each compute tile has an allocated memory partition on the hybrid tile on-chip memory for data writing. The request message array is created by assigning parameter values as mentioned in Figure 4.10. Afterward, the control packet is created by inserting the hybrid tile destination as mentioned in Listing 4.1 (Line:8) followed by the packet payload (Line: 9-10) and transmitting the packet through the NI-TX to the NoC router. Listing 4.2 shows a detailed description of the wait grant message software module executed by a RISC-V core from compute tiles. The compute tile receives a grant packet through the NI-RX as mentioned in (Line: 5-8) Listing 4.2. The compute tile starts to extract the grant packet payload for read, write, or accelerator mode grant as mentioned in Figure 4.10. The grant values are stored in compute tile shared memory in order for every RISC-V

```
1 #define mem_start_address 0x000_1000 // each compute tile has an allocated memory
    partition in hybrid tile for write operations
2 #define mem_end_address   0x000_FFFF
3 uint32_t*const NI_TX = (uint32_t)*0xA000_0000; // NI peripheral address
4 req_msg[32] = {tile_source, mode, w_r, add_read, add_write, size_read, size_write,
    acc_conf, 0, ..., 0}; // add_write+size_write should be within the tile allocated
    partition
5 void hybrid_tile_request(uint_32t req_msg[32], uint_32t msg_size, uint_32t
    hybrid_tile_dest) {
6   uint32_t = 0;
7   while(t < msg_size/32){
8     NI_TX->data = hybrid_tile_dest;
9     for(int m = 0; m < 32; m++){
10      NI_TX->data = req_msg[m+(32*t)];
11    }
12  }
13 return;
14 }
```

Listing 4.1: Hybrid memory/accelerator tile request software module executed on RISC-V cores inside conpute tiles.

```
1 uint32_t*const NI_RX = (uint32_t)*0xA000_F000;// NI peripheral address
2 void wait_grant(uint_32t msg_size, uint_32t read_grant, uint_32t write_grant, uint_32t
    acc_grant) {
3   uint_32t grant_msg[32];
4   for(int t = 0; t < msg_size/32; t++){
5     while(NI_RX->FIFO_data_count < 32);
6     for(int j = 0; j < 32; j++){
7       grant_msg[j+(32+t)] = NI_RX->data;
8     }
9   }
10 read_grant = grant_msg[1];
11 write_grant = grant_msg[2];
12 acc_grant = grant_msg[3];
13 return;}
```

Listing 4.2: Wait grant software module executed on RISC-V cores inside compute tiles.

core to be able to start sending or receiving data to/from the hybrid tile based on the mode of operation. All NIs are considered memory-mapped peripherals from the RISC-V cores inside the compute tile. Accordingly, hybrid memory/accelerator tiles are memory-mapped peripherals to compute tiles in the many-core system. A request message and wait grant software modules can be executed in parallel by compute tile using two RISC-V cores without overlapping due to the dual NI channels for TX and RX. For memory read operations, Listing 4.3 shows a detailed description of the memory read software module executed on RISC-V cores inside compute tiles. The compute tile receives read data packets from the hybrid tile through the NI-RX as mentioned in (Line: 5-6) Listing 4.3. The compute tile starts to extract the data packet payload from the received multiple 32 flits of data (data message)

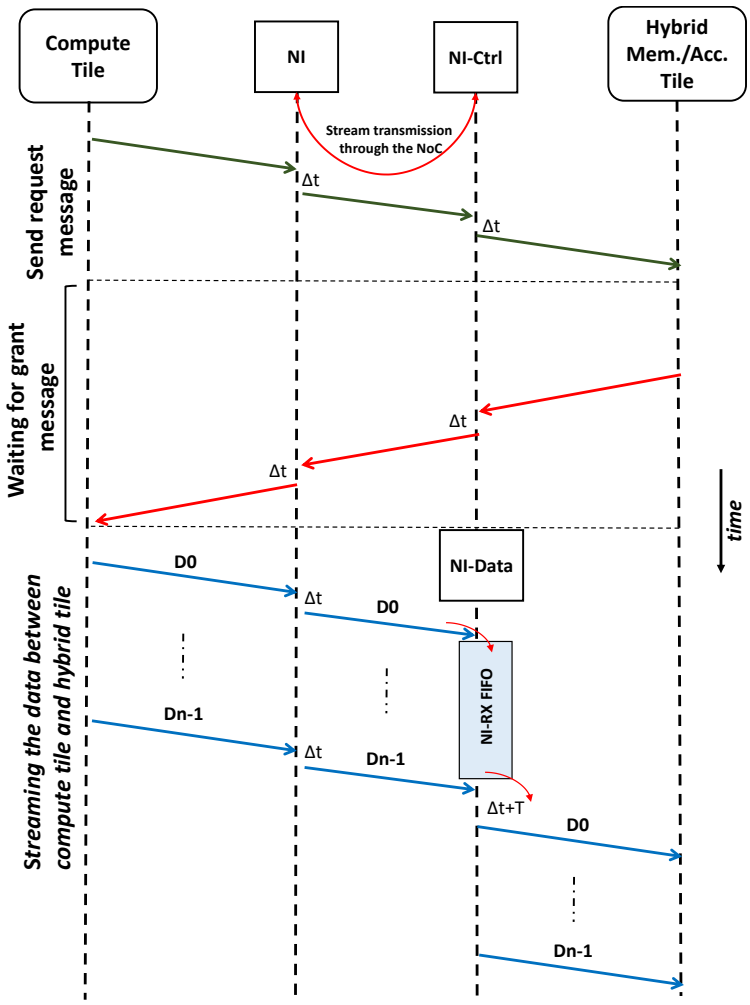


Figure 4.12: Sequence diagram of the data transfer process between a compute tile and hybrid memory tile in case of memory or accelerator data write.

based on the requested memory read data size as mentioned in Figure 4.10. The received data are stored in compute tile shared memory in order for every RISC-V core to be able to access it based on running application requirements. In figure 4.12, a sequence diagram of a memory write operation is presented. Where the compute tile sends a request packet

```
uint32_t*const NI_RX = (uint32_t)*0xA000_F000;// NI peripheral address
void mem_read(uint_32t data_array[data_size], uint_32t data_size) {
  for(int t = 0; t < data_size/32; t++){
    while(NI_RX->FIFO_data_count < 32);
    for(int j = 0; j < 32; j++){
      data_array[j+(32+t)] = NI_RX->data;
    }
  }
return;
}
```

Listing 4.3: Memory read software module executed on RISC-V cores inside compute tiles.

```
1 uint32_t*const NI_TX = (uint32_t)*0xA000_0000; // NI peripheral address
2 void mem_write(uint_32t data_array[data_size], uint_32t data_size, uint_32t
    hybrid_tile_dest) {
3   uint32_t = 0;
4   while(t < data_size/32){
5     NI_TX->data = hybrid_tile_dest;
6     for(int m = 0; m < 32; m++){
7       NI_TX->data = data_array[m+(32*t)];
8     }
9   }
10 return;
11 }
```

Listing 4.4: Memory write software module executed on RISC-V cores inside compute tiles.

through the NoC carrying the request message to the hybrid tile (NI-Ctrl). After a period of time, based on the hybrid tile data path status, a response packet is submitted to the source compute tile carrying the grant message. The grant message informs the compute tile about the readiness of the hybrid tile to respond to the sent request message. In case the grant message informs the compute tile that the hybrid tile can not respond to the submitted request, the compute tile resends again the request message till it receives a grant message to start sending memory write data to the hybrid tile.

For memory write operations, Listing 4.4 shows a detailed description of the memory write software module executed by RISC-V cores inside compute tiles. The compute tile transmits write data packets to the hybrid tile through the NI-RX as mentioned in (Line: 6-7) Listing 4.4. The compute tile inserts the hybrid tile address destination to create the data packet followed by the payload write data message. The number of transmitted data packets is based on the requested memory write data size as mentioned in Figure 4.10. The transmitted data are loaded from the compute tile shared memory to the NI-TX.

Accessing Accelerators from compute tiles

Hardware accelerators can be accessed from compute tiles for both reading and writing data through the NoC in a similar way to memory read/write operations. Hardware accelerator logic can be accessed by configuring the I/O interfaces to be connected to the NI-Data as shown in the tile data path Figure 4.5. Compute tiles use the same reading communication protocol as shown in 4.11 to read data from the hardware accelerator logic through the NoC. Listing 4.5 shows a detailed description of the accelerator read function executed by RISC-V cores inside compute tiles. The compute tile receives read data packets from the accelerator logic through the NI-RX as mentioned in (Line: 5-6) Listing 4.5.

The compute tile starts to extract the data packet payload from the received multiple 32 flits of data (data message) based on the requested accelerator read data size as mentioned in Figure 4.10. The received data are stored in compute tile shared memory in order for every RISC-V core to be able to access it based on running application requirements. On the other hand, regarding writing data to accelerator logic, compute tiles use the same writing communication protocol as shown in 4.12 to write data to the hardware accelerator logic through the NoC. Listing 4.6 shows a detailed description of the accelerator write function executed by RISC-V cores inside compute tiles. The compute tile transmits write

```
1 uint32_t*const NI_RX = (uint32_t)*0xA000_F000;// NI peripheral address
2 void acc_read(uint_32t data_array[data_size], uint_32t data_size) {
3   for(int t = 0; t < data_size/32; t++){
4     while(NI_RX->FIFO_data_count < 32);
5     for(int j = 0; j < 32; j++){
6       data_array[j+(32+t)] = NI_RX->data;
7     }
8   }
9 return;
10 }
11
```

Listing 4.5: Accelerator read software module from the accelerator logic in hybrid tile executed on RISC-V cores inside compute tiles.

```
1 uint32_t*const NI_TX = (uint32_t)*0xA000_0000; // NI peripheral address
2 void acc_write(uint_32t data_array[data_size], uint_32t data_size, uint_32t
    hybrid_tile_dest) {
3 uint32_t = 0;
4   while(t < data_size/32){
5     NI_TX->data = hybrid_tile_dest;
6     for(int m = 0; m < 32; m++){
7       NI_TX->data = data_array[m+(32*t)];
8     }
9   }
10 return;
}1
```

Listing 4.6: Accelerator write software module to the accelerator logic in hybrid tile executed on RISC-V cores inside compute tiles.

data packets to the accelerator logic through the NI-RX as mentioned in (Line: 6-7) Listing 4.6. The compute tile inserts the hybrid tile address destination to create the data packet followed by the payload write data message to accelerator logic. The number of transmitted data packets is based on the requested accelerator write data size as mentioned in Figure 4.10. The transmitted data are loaded from the compute tile shared memory to the NI-TX.

4.3 Evaluation

The hybrid tile architecture is evaluated based on hardware resource utilization and data transfer latency to/from the hybrid memory tile with a set of use cases based on signal processing based kernels and custom hardware accelerators. Two homogeneous tile-based many-core configurations are used for evaluation as mentioned in Section 4.2. Xilinx Virtex Ultrascale+ XCVU9P FPGA device [168] is the target FPGA for implementation and prototyping of the proposed hybrid tile and the target RISC-V-based many-core system. Also, Vivado Design Suite HLx 2019.1 [169] is used for RTL synthesis, simulation, place and routing, and

full and partial bitstream generation. For evaluation, the homogeneous tile-based many-core system and the hybrid memory tile run at a clock frequency of 100 MHz.

4.3.1 FPGA Resource Utilization

The hybrid tile as mentioned in Section 4.1 consists of two main hardware units: the control unit, and the data path unit. Each unit is implemented using a modular and hierarchical design approach. Where internal components are implemented as intellectual property (IP) components to be integrated to build control and data path units. The used on-chip memory is parameterized and its size can be changed during design time based on application requirements and BRAM/URAM availability on the target FPGA. For the current evaluation, 128 BRAM blocks are used with a total size of 512 KiB.

Table 4.2 shows hardware resource utilization of the tile data path components including NI-Data. The tile data path features a low resource utilization overhead in terms of LUTs and FFs. Also, Table 4.3 shows the hardware resource utilization of the tile control unit components. Similar to the tile data path, the control unit features a low resource utilization overhead which makes the hybrid memory/accelerator tile suitable for a wide range of FPGA devices with different sizes and resource counts.

Table 4.4 shows the hardware resource utilization for the tile-based many-core architecture including RISC-V based compute tile, the NoC, and hybrid tiles used for the two many-core configurations as mentioned in Section 4.2. The hybrid tile (without hardware accelerator) consumes $\sim$ 5.5 KLUTs on FPGAs which is considered a low resource utilization in comparison with RISC-V compute tile and NoC resources. Therefore, a scalable number of hybrid tiles to support multiple accelerators or multiple memory banks can be realized with appropriate resource utilization on FPGAs. However, a larger NoC size is required which consumes higher resources. Therefore, a circuit-switched NoC is supported by the target many-core systems which consume fewer resources than a packet-switching NoC. Moreover, for the second configuration considered for evaluation, 30.9%, and 21.9% of the total target FPGA LUTs and BRAMs are consumed respectively.

Table 4.2: Hybrid tile data path resource utilization on Xilinx XCVU9P.

Units	LUTs	FFs	DSPs	BRAMs
Memory Write Manager	203	409	0	0
Memory Read Manager	284	460	0	0
HW Accelerator Wrapper	601	811	0	3
NoC Packet Generator	475	611	0	0
Muxes + Demuxes	372	852	0	0
2xOn-Chip Memory Access	202	389	0	0
NI-Data	162	398	0	9
Total	2382	4075	0	12

Table 4.3: Hybrid tile control unit resource utilization on Xilinx XCVU9P.

Units	LUTs	FFs	DSPs	BRAMs
FSM + Packet Decoding	2892	3319	0	4
NoC Packet Generator	475	611	0	0
NI-Ctrl.	162	398	0	9
Total	2890	3319	0	13

4.3.2 Memory Mode Evaluation

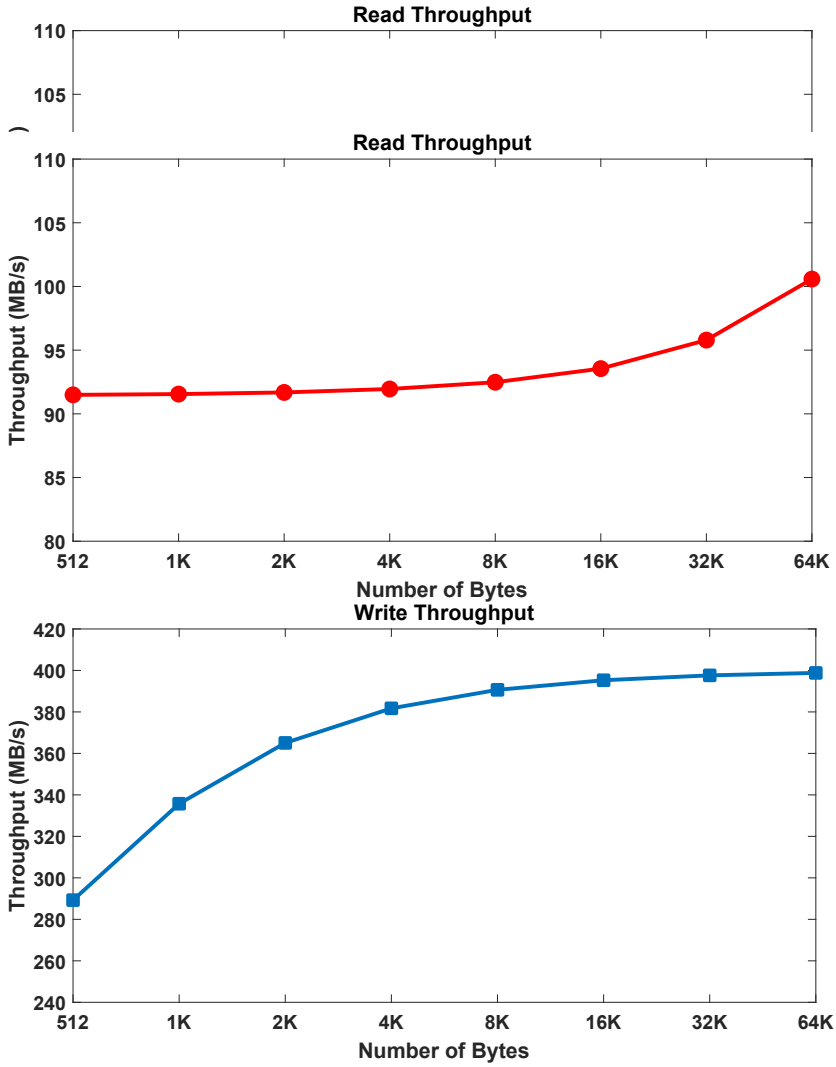
To evaluate the memory mode of the proposed hybrid tile, memory copy functions are executed on RISC-V compute tiles to copy a block of data from the shared compute tile data memory to the hybrid tile (write mode) or vice versa (read mode). Therefore, our evaluation is based on measuring read and write throughput including NoC transfer latency and software kernels overhead of memory copy functions running on compute tiles.

Figure 4.13 shows read and write throughput between a single RISC-V based compute tile and a single hybrid tile for memory mode operation. For memory read mode, a maximum throughput of 100 MB/s is achieved by transferring 64KiB of data from hybrid tile on-chip memory to the shared compute tile data memory as shown in Figure 4.13 (a). The reading throughput increases for large numbers of reading data sizes from the hybrid tile (e.g. 32 KiB, 64 KiB). This is due to the decrease in the ratio between the software time overhead to read the data by any compute tiles and data transmission time over the NoC for large data sizes. In case of a memory write mode, a maximum throughput of 400 MB/s is achieved by copying a block of 64 KiB from the shared data memory of a compute tile to the hybrid tile as shown in Figure 4.13 (b). In the case of a read operation from the hybrid tile, the read data has to pass by the NoC packet generator as in Figure 4.5 to create the data packet to be transferred to the destination compute tile through the NoC. Accordingly, four clock cycles are required to create one packet and send it to the NI-Data. Therefore, the achieved memory write throughput is 4x the memory read throughput. A total maximum data transfer throughput of 500 MB/s is achieved as the hybrid memory/accelerator tile supports simultaneously memory read and write operations using a dual-ported on-chip memory.

Furthermore, several compute and memory-bound kernels are selected for the proposed

Table 4.4: Total resource utilization of many-core configuration-two on Xilinx XCVU9P.

Units	LUTs	FFs	DSPs	BRAMs
Single Compute Tile (32-bit Tile)	36579	15532	24	21
NoC (4x3)	61368	54005	0	0
Single Hybrid Memory/Accelerator Tile	5828	8258	0	153
Total (8xCompute Tile, 2xHybrid Tiles)	365196	194647	192	474
Percentage Utilization on XCVU9P	30.9%	8.23%	2.8%	21.9%

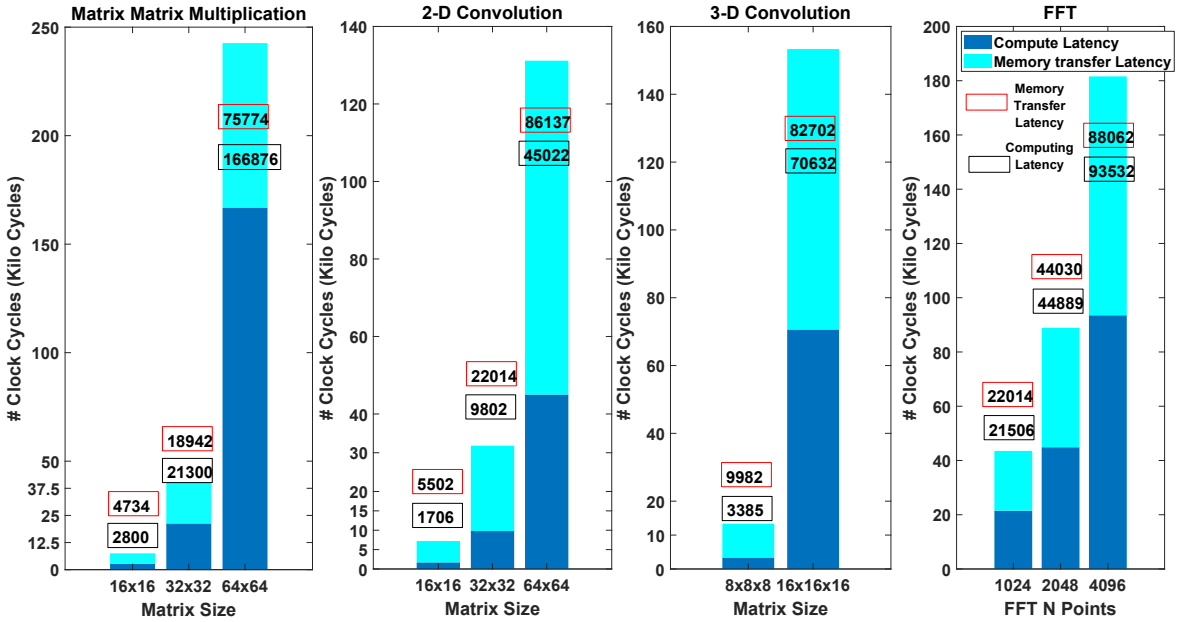


(b) Memory write throughput.

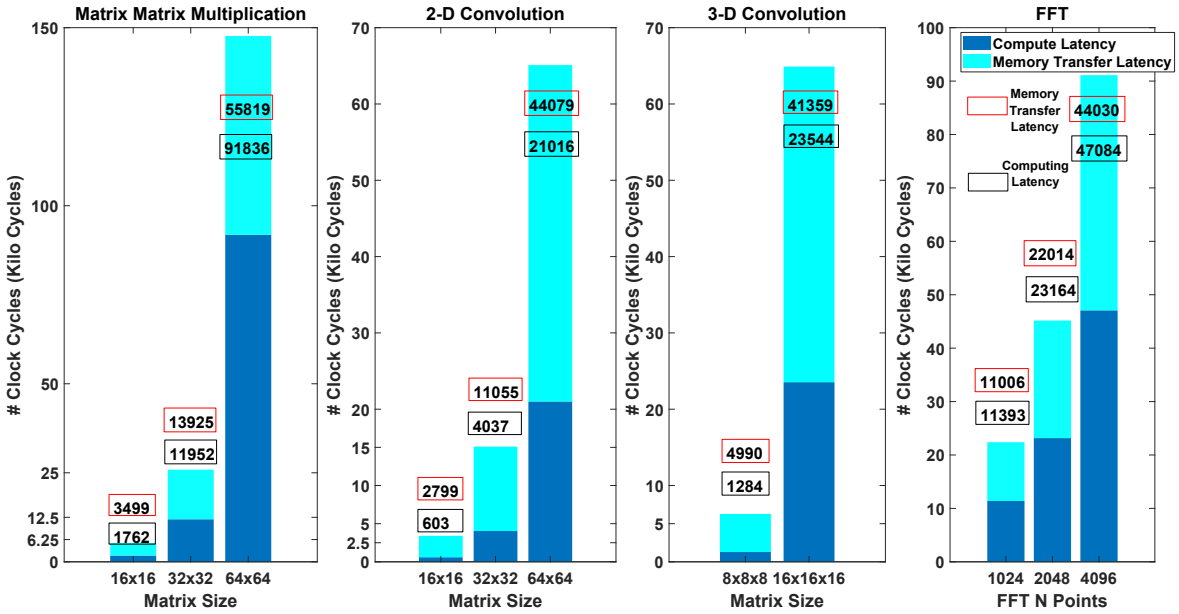
Figure 4.13: Memory bandwidth evaluation between a single compute tile and a single hybrid memory/accelerator tile at a clock frequency = 100 MHz.

system evaluation. The goal of the evaluation is to measure the required compute latency and memory transfer latency to the total execution time on the target RISC-V-based many-core system using the two many-core configurations depicted in Figure 4.9. The selected kernels include matrix multiplication and signal processing software kernels (2D, 3D convolution, FFT) which are executed on the target many-core RISC-V system. The signal processing kernels workloads are parallelized (partitioned) over the specific number of compute and hybrid memory tiles based on the selected many-core configurations. The hybrid memory tile is used as a global shared memory over the NoC between all compute tiles for data sharing. The hybrid tile on-chip memory includes all required data input to be transmitted to all compute tiles upon request.

Initially, all compute tiles read the corresponding input data from the hybrid tile and store them in their shared data memory. During computation, the shared data memory of each tile is used for internal computation by RISC-V cores. Shared data between tiles and final output



(a) Total execution time for application kernels running on the first RISC-V many-core configuration (16xRISC-V cores, 1xhybrid tile).



(b) Total execution time for application kernels running on the second RISC-V many-core configuration (32xRISC-V cores, 2xhybrid tiles).

Figure 4.14: Signal processing based kernels evaluation over tile-based many-core architecture with hybrid memory/accelerator tiles.

data are transferred to the hybrid memory tile. As shown in Figure 4.14, the evaluation is based on total computation time by compute tiles multicore RISC-V and total memory transfer latency to/from the hybrid tile. The total compute time is the required execution time by compute tiles using only their shared data memory to conduct internal computation without

accessing the hybrid tile. The total memory transfer latency includes the total read and write latency by all compute tiles from/to the hybrid tile to/from their shared data memory during signal processing kernels execution. For matrix multiplication kernels ~70% of execution time is consumed by computation. On the other hand, for 2D and 3D convolution ~66% of execution time is consumed by memory transfer. The percentage for computing and memory transfer latency is approximately the same for FFT kernels ~50%. As shown in Figure 4.14 (a), the computation time of all kernels is reduced by ~50% compared to Figure 4.14 (b). On the other hand, memory transfer latency does not scale by the same factor using two hybrid memory tiles for all signal processing kernels due to NoC congestion in case of large size of the transfer data.

4.3.3 Accelerator Mode Evaluation

In order to evaluate the hybrid memory/accelerator tile in the acceleration mode, several custom hardware accelerators are developed and generated using Vivado HLS and integrated inside the accelerator wrapper unit as shown previously in Figure 4.5. All generated hardware accelerators have a single input/output 32-bit AXI-S interface. As shown in Table 4.5, four hardware accelerators from the signal processing domain are developed, generated and synthesized. Those accelerators are: 1) 3D convolution (16x16x16) kernel [89], Xilinx FFT IP (N = 4K points) [173], 3) synthetic aperture radar (SAR) backprojection algorithm [91], and 4) material characterization cost function algorithm [92]. Resource utilization for hardware accelerators is shown in Table 4.5. The first two accelerators (3D convolution, Xilinx FFT IP) are considered small size accelerators with low memory footprint required for PLMs.

On the other hand, the third and fourth accelerators (SAR backprojection, and material characterization cost function) are considered large accelerators size with high memory footprint requirements for PLMs in terms of BRAM blocks. For evaluation, a compute tile sends a request packet to configure the hybrid tile to the accelerator mode and set the accelerator wrapper interface to be connected to the on-chip memory for read and write data. The request packet contains the start address for input and output data in the memory as well as the specific size of data to be stored or loaded from the PLM. Similar to our evaluation for memory mode, compute and memory transfer latency are measured for all four hardware accelerators as shown in Figure 4.15.

Accordingly, memory latency is the data transfer time between accelerator logic and on-chip memory in the hybrid tile. It indicates accelerators memory bound degree. For example, the 3D convolution consumes more compute time than memory latency therefore it is a compute-bound application compared to Xilinx FFT IP which is a memory-bound application

Table 4.5: Hardware accelerator resource utilization on Xilinx XCVU9P.

Hardware Accelerator Kernel	LUTs	FFs	DSPs	BRAMs
3D Convolution (16x16x16)	1099	1026	9	3
Xilinx FFT IP (N = 4K points)	3135	4984	15	6
SAR Backprojection Function	11527	9930	24	150
Material Characterization Cost Function	37872	31589	219	45

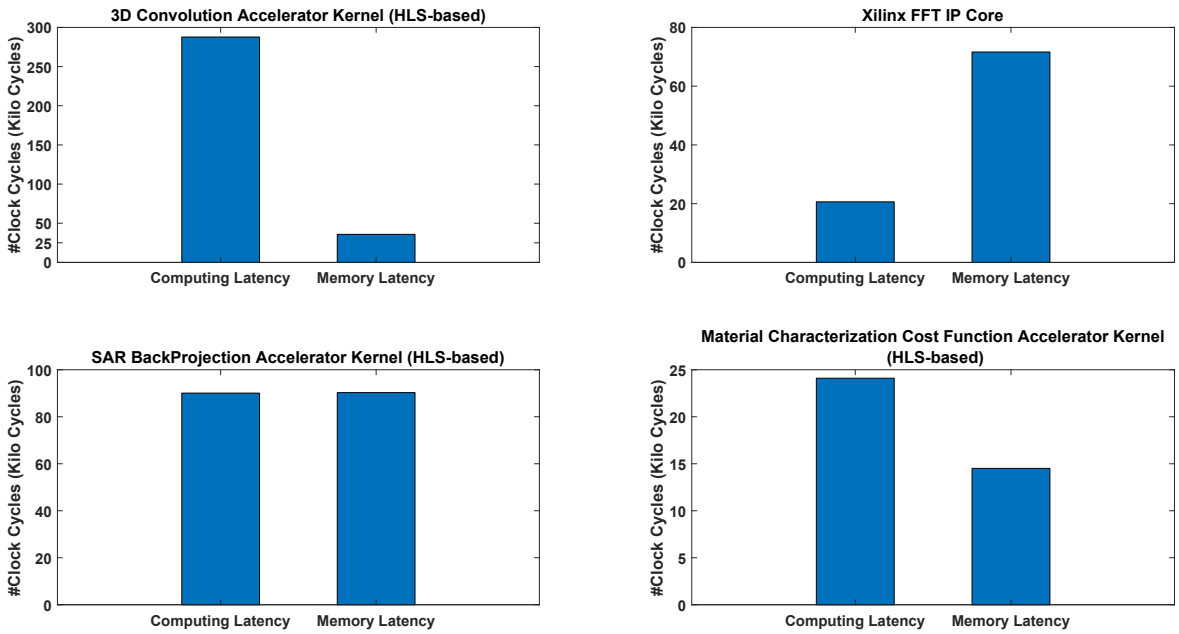


Figure 4.15: Hardware accelerator performance evaluation.

as shown in Figure 4.15. On the other hand, the SAR backprojection algorithm is a compute and memory-bound application that requires a large memory size and long memory latency for processing large data sizes (360KB of data). Therefore, the hybrid tile supports both compute and memory-bound accelerators with large data sets.

4.4 Summary

Chapter 4 presents a novel hybrid memory/accelerator tile architecture for tile-based many-core systems. The hybrid tile supports two modes of operations which are selected during run-time as a memory or an accelerator tile. The hybrid tile is based on a modular and reusable architecture implementation that can be seamlessly integrated into a tile-based many-core system. Tile controlling and configuration are conducted through a message-based communication protocol. Where control signals and data are carried through the NoC using control and data messages. The proposed tile is evaluated using two tile-based many-core configurations with different numbers of RISC-V cores and multiple instants of hybrid memory/accelerator tiles. The evaluation shows low hardware resource requirement per single hybrid tile and a maximum data transfer of 500 MB/s between a single compute tile and a single hybrid tile.

Section 4.1 presents the hybrid tile architecture implementation for FPGA-based many-core systems. The hybrid tile provides the flexibility to seamlessly integrate LCAs into tile-based many-core architecture. In addition, during accelerator inactive time, the hybrid tile allows the re-use of accelerator PLM as a shared scratchpad memory between many-core compute tiles. Therefore, the hybrid tile allows efficient reuse of FPGA on-chip memory based on application requirements at runtime. The hybrid tile consists of two main architectural parts. The first part is the tile data path which is responsible to establish several data paths between the hybrid tile NI-Data and on-chip memory based on the requested mode of operation.

Three data paths for memory read, write, and acceleration modes can be established based on the received request messages from compute tiles. The tile data path is controlled by the hybrid tile control unit which is the second architectural component. The tile control unit is responsible to receive and transmit request and response messages from/to compute tiles. It decodes the received control packets through the NoC to extract the request messages in order to identify the mode of operation. In addition, the control unit has a FSM that manages and controls the tile data path by sending and receiving control and configuration signals to ensure proper functionality of the tile data path and monitors its status.

Section 4.2 presents the integration between the hybrid memory/accelerator tile and the tile-based many-core systems. A message-based communication protocol is developed to manage and control the data transfer between the hybrid tile and compute tiles through the NoC. The hybrid tile is integrated to the many-core NoC through two NIs using two routers one for data and the second for control signals to avoid control and data packets overlapping during operations. LCAs with AXI-S interfaces are seamlessly integrated into the hardware accelerator wrapper inside the tile data path. The hybrid tile only supports a single LCA. Several software modules for sending/receiving control packets, and read and write operations to/from the hybrid tile are developed based on a message-based communication protocol. Software modules are executed from RISC-V cores inside compute tiles using the shared memory to load/store control and data packets to NIs. Two tile-based many-core configurations are realized for evaluation. The first configuration consists of 4xcompute tiles (16xRISC-V cores) with a single hybrid tile. The second configuration consists of 8xcompute tiles (32xRISC-V cores) and two hybrid tiles.

Finally, Section 4.3 presents the evaluation and obtained experimental results of the hybrid memory accelerator tile using several use cases based on signal processing kernels and hardware accelerators for performance evaluation in terms of memory transfer latency and computing time for two tile-based many-core configurations. The hybrid tile features low resource utilization in terms of LUTs and FFs which make it suitable for a wide range of FPGA sizes. The hybrid tile provides a memory bandwidth of 100 MB/s to a single compute tile for read operation. For the memory write operation, a memory bandwidth of 400 MB/s is achieved between a single compute tile and a single hybrid tile. Overall, a maximum memory bandwidth of 500 MB/s is achieved by a single hybrid tile for simultaneous read and write operations. Several hardware accelerator kernels from the signal-processing domain are used to evaluate the hybrid tile acceleration mode. The evaluation is based on accelerator logic computing time and memory transfer latency between on-chip memory (PLM) and accelerator logic.

5 Reconfiguration Management for Self-Adaptive RISC-V based Many-Core Architectures

Nowadays edge and near-sensor applications are creating new challenges and motivation toward new computing paradigms based on openness and flexibility as well as energy efficiency and high performance. Hence, the focus is shifted toward the development of agile hardware platforms to offer the required computing performance in parallel with modularity, adaptability and openness. Therefore, FPGAs by their adaptability and reconfigurability features along with the modularity and open-source characteristics of RISC-V ISA lay the motivation towards the realization of agile hardware platforms.

Several state-of-the-art platforms combine a RISC-V processor (either as a soft or hard core) with an FPGA fabric on the same chip. For IoT and industrial applications, Microchip Polarfire SoC [174] tightly couples a quad-core RISC-V-based processor supporting a Linux operating system with a low-power FPGA fabric on a single chip. The Microchip Polarfire platform can provide offloading of application tasks from the programmable RISC-V-based processor to the FPGA fabric during run time where the FPGA fabric hosts several hardware accelerators based on the target application domain. Moreover, Flexbex [175] and Arnold [176] provide the extension of a RISC-V-based microcontroller unit with an embedded FPGA (eFPGA) for ultra-low power devices suitable for near-sensor and embedded computing. However, the above-mentioned RISC-V-based FPGA platforms lack the capability of DPR to exchange selected kernels of the running application on the FPGA fabric without halting the whole application.

Furthermore, those RISC-V-based FPGA platforms are designed for low-power applications and are not suitable for dynamic and high-workload applications due to the limited size of the eFPGA and the high development cost. Therefore, high-end FPGAs such as Xilinx or Intel devices are considered the most suitable platforms for dynamic and high workloads applications. Those high-end FPGAs support the DPR feature to enable the fragmentation of the FPGA fabric into multiple isolated partitions with different sizes to host multiple hardware modules that can be swapped during runtime without halting other hardware modules on the static region. Also, they feature fast prototyping for a wide range of SoC implementations hosting several types of soft-core processors with multiple hardware-accelerated modules.

In this chapter, a novel reconfiguration management approach dedicated to FPGA-based RISC-V SoC is developed and implemented. The proposed approach extends the FPGA-based RISC-V SoCs to support DPR for their tightly coupled reconfigurable accelerators for more

flexibility and dynamic workload. The proposed reconfiguration management approach consists of a hardware-implemented part called (RV-CAP) for DPR controlling on the FPGA fabric and a set of software drivers and application programming interfaces (APIs) to manage the DPR process from a programmable software environment from a RISC-V core [21].

Existing research platforms do not fully meet modularity and self-adaptability requirements for heterogeneous RISC-V-based multi-/many-core systems. In Chapter 3, a modular tile-based architecture for multi RISC-V ISAs is presented with several heterogeneous types of compute tiles that can be configured during design time. In this Chapter, a novel self-adaptive tile-based many-core architecture for heterogeneous RISC-V-based many-core configurations targeting FPGA devices is presented targeting FPGA devices. The proposed architecture satisfies the requirements of self-adaptability and modularity for highly scalable heterogeneous RISC-V-based many-core systems on FPGAs.

The reconfiguration management unit is integrated within the main processing tile of the tile-based many-core architecture in order to support the adaptability feature for the proposed architecture by self-managing the DPR process from the main processing tile to change the type of compute tile of the tile-based architecture at run-time.

The Chapter is structured as follows. Section 5.1 presents the internal DPR management unit for self-adaptive RISC-V-based SoC. The DPR management unit is developed based on a hardware reconfiguration management unit called RV-CAP suitable for RISC-V-based SoC. In addition, the RV-CAP is integrated first on a single core RISC-V-based SoC for development and evaluation. Section 5.2 presents the set of APIs and the software abstraction layer to control and manage the RV-CAP unit from the RISC-V core. Moreover, the proposed reconfiguration approach supports the integration of DPR vendor controllers such as Xilinx AXI-HWICAP [177] through a custom set of APIs to control it from the RISC-V core. Section 5.3 presents the evaluation of the reconfiguration management unit in terms of required resource utilization by the hardware-based unit (RV-CAP), and the achievable reconfiguration time by using both the RV-CAP and Xilinx AXI-HWICAP controllers. In addition, a set of image processing-based accelerators is used as reconfigurable modules (RM) targeting a single reconfigurable partition (RP) within the RISC-V-based SoC for evaluation. Section 5.4 presents the integration of the proposed reconfiguration management unit into the tile-based many-core architecture. Finally, Section 5.5 summarizes this chapter.

5.1 Internal Dynamic Partial Reconfiguration Management for Self-Adaptive RISC-V based SoC

This section presents an internal DPR manager suitable for FPGA-based RISC-V SoC that offers a hardware/software management of the reconfiguration process at run-time. The DPR manager is targeting high-end FPGA devices, in our case we are targeting Xilinx FPGA devices that feature the DPR capability. The DPR manager consists of an internal DPR controller called RV-CAP responsible for low-level control of the DPR process over the FPGA fabric and loosely coupled with the RISC-V soft-core within the SoC. In addition, a set of software drivers and APIs to manage the reconfiguration process from the RISC-V core within a programmable software environment. Moreover, the selected FPGA-based RISC-V SoC is equipped with a 64-bit RISC-V application class processor compatible with the RV64GC ISA (Ariane core) similar to the RISC-V core used by the main processing tile for the tile-based architecture

presented in Chapter 3. Hence, the chosen SoC is capable of executing medium-workload applications with multiple coupled hardware accelerators for different hardware-accelerated tasks. In this section, the target FPGA-based RISC-V SoC is described first followed by a detailed description of the RV-CAP controller architecture.

5.1.1 FPGA-based RISC-V SoC

An open-source RISC-V SoC architecture (CVA6) [78] based on a 64-bit, single-issue, in-order RISC-V core (RV64IMAC Ariane) is used for the development of the proposed internal DPR manager. The RV64IMAC ISA supports multiply/divide, and atomic memory operations besides integer operations. Therefore, the used Ariane core configuration complies with the RISC-V 64-bit ISA and supports variable compressed instructions length (C). Besides, hardware implementation of multiplication and division units within the execution stage. Thus, Ariane has been chosen for its characteristic as an application class processor suitable for medium-workload SoC implementations.

Figure 5.1 shows a schematic overview of the target FPGA-based RISC-V SoC. The SoC is using a bus-based architecture based on a 64-bit AXI-4 interconnect where the Ariane core, SoC peripherals and hardware accelerators are communicating through a shared bus. The Ariane core is the master unit on the AXI-4 interconnect, while the rest of the peripherals are slave memory-mapped components within the core's address space ranges. In addition, an on-chip boot memory is used to store the application instructions for execution. Moreover,

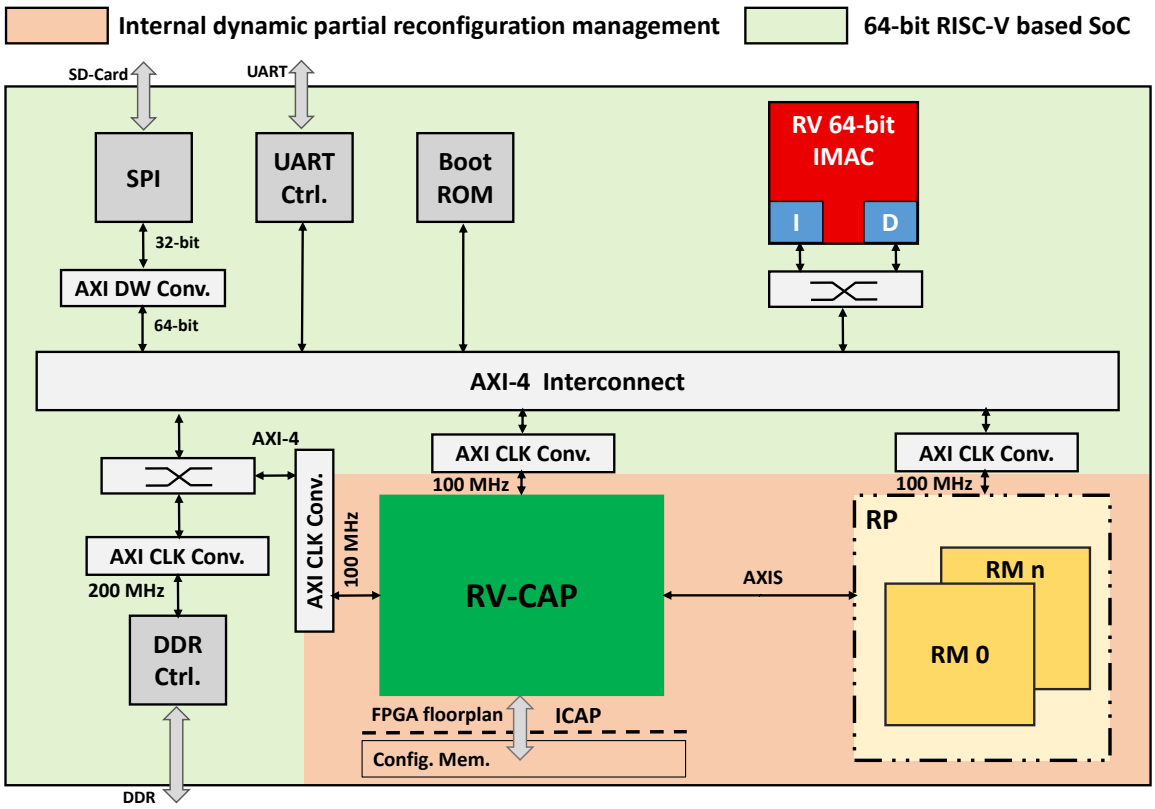


Figure 5.1: A schematic overview of the target self-adaptive RISC-V based SoC [21].

the AXI-4 interconnect allows tightly coupled integration of several hardware accelerators to the Ariane core. Therefore, one or multiple RPs can be created for hosting different RMs of hardware accelerators.

The SoC uses a set of open-source non-coherent on-chip communication components for AXI data-width converter, and AXI clock converters [165] that are useful for the integration of different peripherals operating with different clock frequencies and supporting different data widths (e.g. DDR, UART, SPI controllers). In addition, the integration of the proposed internal DPR manager requires the insertion of an additional AXI clock converter between the main AXI-4 bus and the DDR memory controller as the DPR manager is operating at a clock frequency of 100 MHz suitable for the hard-core ICAP interface on the FPGA floorplan. Also, optional AXI clock converters and AXI data width converters can be inserted between the hosted hardware accelerators by RPs and the main AXI-4 bus.

Furthermore, a set of software drivers to access the SoC I/O peripherals have been developed in order to load the partial bitstreams from an external SD-card to the SoC DDR memory. In addition, a set of utility modules to communicate with memory-mapped peripherals for reading and writing data through the core address space range. For reading and writing logical blocks from the SD-card, the serial peripheral interface (SPI) is used to communicate between the AXI-4 bus and the external SD-card. Therefore, a set of file I/O software functions is developed based on the minimalist file allocation table (FAT32) implementation to support reading, writing, and overwriting of partial bitstream files. Afterwards, a set of software timer modules are created to count the clock cycles to measure the reconfiguration time of different partial bitstream sizes. For RISC-V core programming, the RISC-V GNU compiler toolchain [167] is used to compile the application source codes (i.e., *C* codes) for the RV6IMAC architecture.

5.1.2 DPR Controlling Unit (RV-CAP)

The RV-CAP controller for the proposed DPR manager is shown in Figure 5.2. The RV-CAP architecture features a high throughput data transfer rate to the FPGA configuration memory through the hard-core ICAP primitive during DPR reconfiguration process to achieve a small reconfiguration time for large FPGA partition sizes. In addition, auxiliary functions related to direct data streaming between the DDR and hosted hardware accelerator modules are supported. This is only in the case of loosely coupled accelerators directly attached to the internal SoC AXI interconnect. Therefore, the RV-CAP controller supports two modes of operation described as follows:

- DPR mode of operation: In this mode, the RV-CAP is used to manage partial bit stream data transfer between the external DDR and the internal configuration memory of the FPGA through the ICAP primitive. The RV-CAP acts as a custom internal ICAP controller for managing the DPR process.
- Hardware accelerator mode of operation: In this mode, the RV-CAP is used to manage stream data transfer from/to hardware accelerator modules hosted by a single RP and the external DDR memory as shown in Figure 5.1.

The RV-CAP controller has four interfaces to communicate and for data transfer with the RISC-V-based SoC. Two 64-bit AXI interfaces to the main AXI-4 crossbar for controlling the DMA controller, and R/W control signals to the configuration mode of operation interface

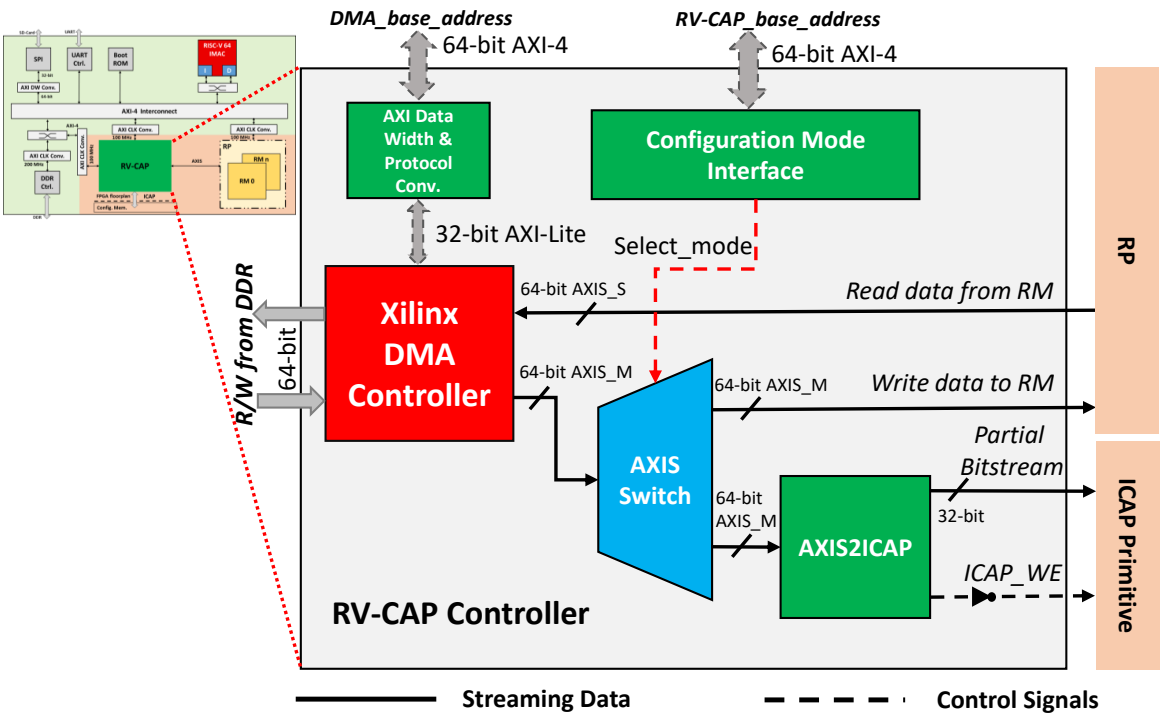


Figure 5.2: Overview of the RV-CAP controller architecture [21].

as shown in Figure 5.2. In addition, two interfaces for the partial bitstream and RM stream data transfer from the external DDR to the ICAP primitive or the RM based on the RV-CAP mode of operation. The RV-CAP is fully controlled by the RISC-V core as a memory-mapped slave peripheral. On the other hand, the RV-CAP is a master unit when it interacts with the DDR controller through the DMA for read and write stream data. The RV-CAP is working at a single clock source (100 MHz) in a fully synchronized design. Therefore, the RISC-V-based SoC has three clock domains as shown in Figure 5.1 for the DDR controller, the internal DPR manager, and the rest of the system. The RV-CAP internal architecture consists of the following architectural components described as follows:

1. A Xilinx DMA controller IP [178] connected to the SoC DDR controller through an additional crossbar as shown in Figure 5.1. Thus, the DMA controller is a master component to the DDR controller as well as the RISC-V processor. The DMA is configured to transfer a 64-bit data word from the RISC-V-based SoC DDR memory to either the ICAP interface or the RM based on the selected mode of operation. The DMA blocking mode is selected during partial bitstream loading to the FPGA configuration memory.

2. AXI data width and protocol converters are used to convert between the 64-bit AXI-4 standard of the main 64-bit AXI interconnect to the 32-bit AXI-lite control interface of the Xilinx DMA.

3. A configuration mode interface is implemented to R/W controlling signals to select the RV-CAP mode of operation either the DPR mode or the hardware accelerator mode.

4. An AXI-stream switch is inserted between the Xilinx DMA and the ICAP and RM data input interfaces to select whether the RV-CAP controller is operating on the DPR mode

or the accelerator mode by connecting the DMA data stream interfaces to the RM or the ICAP primitive.

5. An AXI-stream to ICAP converter is implemented to connect the AXI-stream data to the ICAP data port and convert between the 64-bit data stream to a 32-bit data stream compatible with the ICAP primitive input data width. The AXIS2ICAP block shown in Figure 5.2 is responsible to convert a data word of 64-bit fetched from the DDR memory into two 32-bit data words, which are written in order to the ICAP data port. Also, the valid stream signal is inverted and connected to the *icap_we port*. as the ICAP is operating in the writing mode into the configuration memory. The read/write select input port is permanently set to zero.

5.2 Application Programming Interfaces (APIs) and Abstraction Layer

The reconfiguration process is done through three steps in order to successfully load a new partial bitstream with a new RM into the RP. It starts by first loading the RMs partial bitstream files from the SD-Card to specific DDR memory addresses. Where each RM is stored on a specific partition on the DDR memory. Second, the RV-CAP is configured to operate in the DPR mode of operation. Afterwards, the reconfiguration process starts by reading the partial bitstream of a specific RM from the DDR and load it to the FPGA configuration memory through the RV-CAP controller. The three steps are fully managed and controlled by the RISC-V core through a set of APIs creating a software abstraction layer to manage and control the reconfiguration process from a software programmable environment. This section presents and describes the RV-CAP APIs to manage and abstract the reconfiguration process from the RISC-V core. In addition, the potential to support other DPR vendor controllers such as AXI HWICAP [177] is elaborated.

5.2.1 RV-CAP APIs

The interaction between the RV-CAP unit and RISC-V core is conducted through a set of API software modules. Where the RISC-V core manages the reconfiguration process by selecting the desired RMs to be loaded based on application requirements. The aforementioned three steps of the reconfiguration process are fully managed through software modules running on the RISC-V core.

As shown in Listing 5.1, the first step (line:2) is initializing all RMs by reading all partial bitstream files from the SD-Card and storing them on the external DDR memory. The second and third

```
1 // initializing all RMs and load partial stream from SD-Card to the SoC DDR
2 init_RM (RM, RM_number, pbit_fat_partition);
3 // initialize RV-CAP reconfiguration process
4 init_reconfig_process (RM, config_mode);
```

Listing 5.1: RM initialization and reconfiguration process API software modules to control the RV-CAP from RISC-V core.

steps (line:4) are the selection of mode configuration to be the DPR mode of operation and afterwards start the reconfiguration process. In more detail, Listing 5.2 shows a detailed description of the RMs initialization modules. Each RM has a specific size and start address to store on the external DDR memory as mentioned in Listing 5.2 (line:1-5) *pr_module*. the process starts by reading a specific RM partial bit file using the FAT file systems (line:16) from the SD-Card and storing it to the RM location on the DDR (*RM[i].start_adr, RM[i].byte_size*). The RM initialization software module is responsible to load a specific number of RMs for different RPs. Therefore, the RM initialization software module is suitable for multi-partition design.

Listing 5.3 shows a detailed description of the RV-CAP reconfiguration process software module. After loading all RMs for all existing RPs in the design, the RISC-V core has to select which RM partial bit file is needed to be active. The reconfiguration process can only reconfigure a single RP with a single RM at a time. Therefore, if multiple RPs have to be reconfigured, the reconfiguration process has to be repeated sequentially over the number of selected RMs. As shown in Listing 5.3 (line:2-3), the reconfiguration process starts by reading the start address and the size of the partial bit file of the selected RM. Afterwards, the RV-CAP mode of operation is set to the DPR mode of operation (line:4).

The next step as shown in Listing 5.3 (line:5) is the Xilinx DMA initialization (*DMA_start*) to start the Xilinx DMA reading channel for data streaming between the DDR and the ICAP primitive to load the RM partial bit file to the configuration memory. The DMA starts to write the data from the external DDR to the AXIS2ICAP block in Figure 5.2 through the DMA write stream

```
1  typedef struct{
2      char fname[13]; // filename on SD card
3      uint8_t *start_adr; // DDR start address on the DDR
4      uint32_t byte_size; // partial bitstream size
5    } pr_module;
6
7  typedef struct{ // fat partition parameters for loading partial bitstream files from
   the SD-Card
8      uint32_t fat_begin ;
9      ....
10     uint32_t cluster_loaded ;
11   } fat_partition;
12 // RM initialization software function
13 void init_RM (pr_module *RM, uint32_t RM_number, fat_partition *pbit_fat_partition){
14     uint32_t pr_module_data[pr_mod_data_size];
15     for(uint32_t i=0; i<RM_number; i++){
16         uint32_t bit_file = open_file(RM[i].fname, pbit_fat_partition);
17         uint8_t *start_adrress = pr_module_data + i * pr_mod_data_size;
18         uint32_t bytes_read = read_file(start_adrress, bit_file.size); // read a
   partial bit file from the SD-Card and store it on a specific location on the DDR
19         RM[i].start_adr = start_adr;
20         RM[i].byte_size = bytes_read;
21     }
22 return 0;
23 }
```

Listing 5.2: An overview of the RM initialization API software module.

```
1 void init_reconfig_process (pr_module *RM, uint32_t config_mode){
2     uint32_t start_address = RM->start_adr;
3     uint32_t byte_size = RM->byte_size;
4     write_reg(RV_CAP_BASE, config_mode); // select RV-CAP mode of operation to be DPR
5     dma_start(); // start DMA
6     dma_write_stream_start(start_address, byte_size); // start data transfer from the DDR
    to the RV-CAP
7 return 0;
8 }
```

Listing 5.3: An overview of the RV-CAP reconfiguration process API software module.

module as shown in Listing 5.3 (line:6). Listing 5.4 shows a detailed description of the DMA write module. The module is responsible to configure the Xilinx DMA IP through the DMA control interface (*DMA_base_address*) in Figure 5.2. The DMA configuration is conducted by setting the DMA control register to work in the normal mode of operation as shown in Listing 5.4 (line:5), and setting the start address and data transfer size (line:6-7).

5.2.2 Supporting DPR Vendor Controller

Xilinx provides several hardware solutions that enable an embedded soft-core processor to manage and control the DPR process through a set of software modules. Among those solutions, Xilinx offers an IP core called AXI HWICAP. In a conventional way, a master soft-core processor or a control unit (e.g., MicroBlaze) is used to manage the transmission of partial bitstreams from a storage unit (e.g., DDR) to the ICAP primitive on the FPGA floorplan to load the configuration memory.

In this subsection, an alternative DPR controller unit (AXI HWICAP) is used to allow the RISC-V core to take over the task of the master unit to manage the writing process to the FPGA configuration memory through the ICAP primitive. Some modifications have been done to integrate the AXI HWICAP IP core into the RISC-V based SoC. On the hardware side, we add a data width converter (from 64-bit to 32-bit) as well as a protocol converter (from AXI4 to AXI4-Lite) to match the IP core AXI4-Lite slave specifications.

```
1  #define DMA_MM2S_CR (DMA_BASE + 0x00) // DMA control register base address
2  #define DMA_MM2S_SA (DMA_BASE + 0x18) // DMA source address register base address
3  #define DMA_MM2S_LENGTH (DMA_BASE + 0x28) // DMA transfer length register base address
4  void dma_write_stream_start (uint32_t* start_address, uint32_t byte_size){
5      write_reg(DMA_MM2S_CR, 0); // set the DMA control register to run on the normal
   mode
6      write_reg(DMA_MM2S_SA, start_address); // set the DMA start address from the DDR
7      write_reg(DMA_MM2S_LENGTH, byte_size); // set the DMA data stream size to read from
    the DDR
8  return 0;
9  }
```

Listing 5.4: An overview of the DMA write API software module.

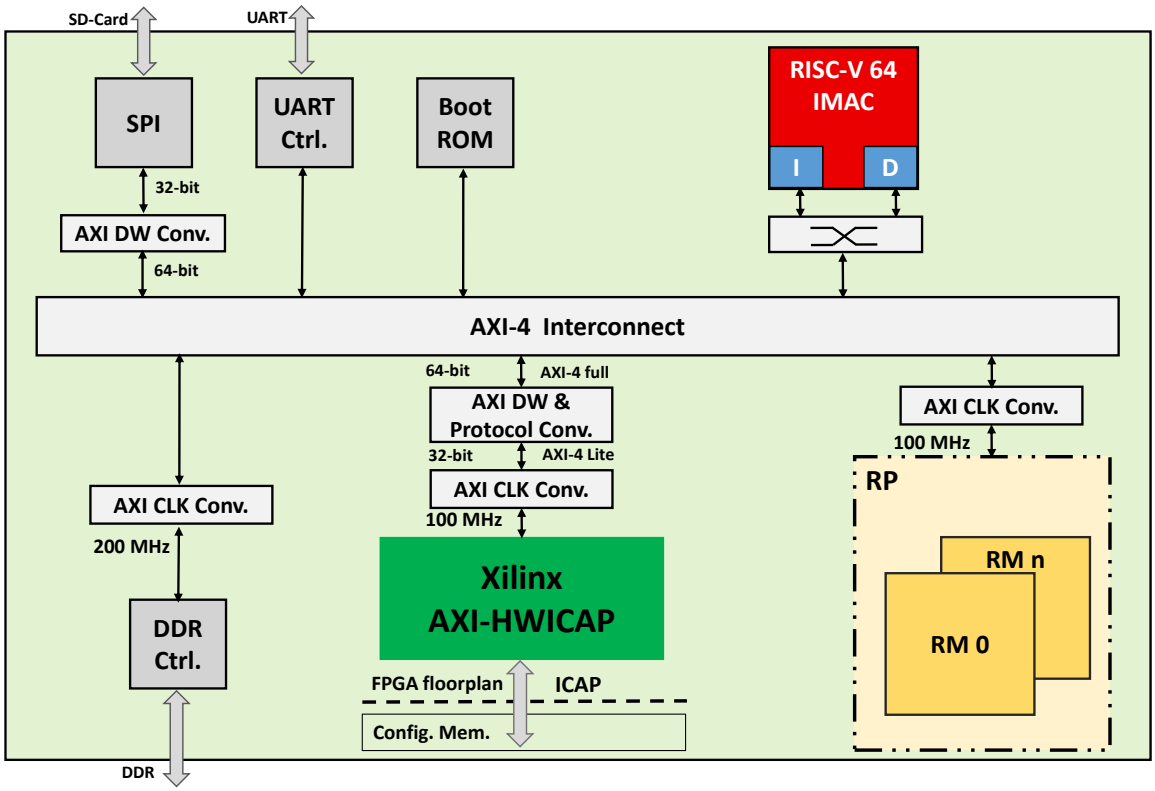


Figure 5.3: A schematic overview of the target self-adaptive RISC-V based SoC with Xilinx AXI-HWICAP controller.

Figure 5.3 shows a schematic overview of the RISC-V-based SoC with the Xilinx AXI HWICAP controller with the extra AXI data width and clock converters needed for the integration. The AXI HWICAP is also running on the same clock frequency of 100 MHz similar to the operating frequency of the RV-CAP controller. Furthermore, we re-sized the internal write FIFO of the AXI HWICAP module to 1024 locations to improve the time transfer during data transfer between the DDR and the ICAP interface through the AXI HWICAP internal FIFO. Similar to RV-CAP controller, to allow the AXI HWICAP to be controlled by the RISC-V core, a set of API modules has been developed to enable partial reconfiguration through the ICAP.

Similar to the RV-CAP controller, As shown in Listing 5.5, the reconfiguration process is conducted through multiple steps. In the case of AXI HWICAP only two steps are required. The first step as shown in Listing 5.5 (line:2) is initializing all RMs by reading all partial bitstream files from the SD-Card and storing them on the external DDR memory. The second step (line:4) is the reconfiguration process by configuring the AXI HWICAP to start receiving

```
1  // initializing all RMs and load partial stream from SD-Card to the SoC DDR
2  init_RM (RM, RM_number, pbit_fat_partition);
3  // initialize reconfiguration process through AXI-HWICAP
4  init_reconfig_process_hw_icap (RM);
```

Listing 5.5: RM initialization and reconfiguration process API software modules to control the Xilinx AXI-HWICAP from RISC-V core.

```
1  void init_reconfig_process_hw_icap (pr_module *RM){
2      init_hw_icap(); // initialize the AXI HWICAP
3      uint32_t start_address = RM->start_adr;
4      uint32_t byte_size = RM->byte_size;
5      hw_icap_write(start_address,byte_size); // start writing partial bit stream from
   the DDR to the AXI HWICAP
6      hw_icap_done(); // check for AXI HWICAP done process for successful transfer of
   partial bit file
7  return 0;
8  }
```

Listing 5.6: An overview of the AXI-HWICAP reconfiguration process API software module.

the partial bitstream data from the DDR through the AXI interconnect and write it to the configuration memory through the ICAP interface.

Listing 5.6 shows a detailed description of the AXI HWICAP reconfiguration process software module. After loading all RMs for all existing RPs in the DDR, the RISC-V core has to select which RM partial bit file is needed to be active. The reconfiguration process can only reconfigure a single RP with a single RM at a time. Therefore, if multiple RPs have to be reconfigured, the reconfiguration process has to be repeated sequentially over the number of selected RMs. As shown in Listing 5.6 (line:2-4), the reconfiguration process starts by initializing the AXI HWICAP controller followed by reading the start address and the size of the partial bit file of the selected RM. Afterwards, the AXI HWICAP starts receiving the partial bit files from the DDR through the AXI interconnect and writes them to the configuration memory as shown in Listing 5.6 (line:5). Finally, the module checks for the successful transfer of all partial bitstream payloads by checking the AXI HWICAP done signal (line:6).

Listing 5.7 shows a detailed description of AXI HWICAP write software module. The module is responsible to configure the AXI HWICAP IP through the 32-bit AXI Lite control interface. The AXI HWICAP configuration is conducted by setting the AXI HWICAP control register to work in the write mode of operation as shown in Listing 5.7 (line:6). The RISC-V core loads the partial bitstream data from the DDR and sends them in 4-Byte words to the AXI HWICAP (line:5). The RISC-V core writes the partial bitstream data to the AXI HWICAP internal FIFO. The filling and flushing of the internal write FIFO is repeated until the complete partial bitstream has been transferred.

```
1 #define ICAP_WF (ICAP_BASE + 0x100) // write FIFO keyhole register base address
2 #define ICAP_CR (ICAP_BASE + 0x10C) // write FIFO control register base address
3 void hw_icap_write (uint32_t *start_address, uint32_t byte_size){
4     for(uint32_t i=0; i<byte_size; i++){
5         write_reg(ICAP_WF, *start_address++); // write partial bitstream data to the AXI
   HWICAP internal FIFO
6         write_reg(ICAP_CR, 0x01); // set the AXI HWICAP on the write mode
7     }
8 return 0;
9 }
```

Listing 5.7: An overview of the Xilinx AXI-HWICAP write API software module.

5.3 Evaluation of the Reconfiguration Management Approach

The reconfiguration management approach including the RV-CAP controller is evaluated based on hardware resource utilization and achievable reconfiguration time targeting the self-adaptive FPGA-based RISC-V SoC presented in Section 5.1 as shown in Figure 5.1. During development and implementation, the Xilinx Virtex Ultrascale+ XCVU9P FPGA device [168] is the target FPGA for implementation and prototyping of the proposed reconfiguration management approach and the target RISC-V-based SoC. Also, Vivado Design Suite HLx 2019.1 [169] is used for RTL synthesis, simulation, place and routing, and full and partial bitstream generation. For evaluation, the RISC-V-based SoC is running at a clock frequency of 120 MHz and the RV-CAP controller at a clock frequency of 100 MHz. The APIs software modules are developed using C language and the RISC-V GNU compiler toolchain [167] is used to generate executable code for the RV64IMAC core.

5.3.1 Hardware Resource Evaluation

The hardware part of the reconfiguration management is the RV-CAP controller or the Xilinx AXI HWICAP controller. As mentioned in Section 5.1, the RV-CAP controller (see Section 5.2) consists of several hardware components including AXI-based modules for interfacing with the RISC-V-based SoC and a single Xilinx DMA IP. Table 5.1 presents the hardware resource utilization of the RV-CAP controller and the Xilinx AXI HWICAP controller. The maximum AXI burst size of the Xilinx DMA controller is set to 16. As mentioned in Section 5.1, all components of the RISC-V-based SoC communicate over a bus width of 64 bits with the exception of the AXI HWICAP and the control signals, which have a data width of 32 bits. As we can see from Table 5.1, the AXI HWICAP utilizes fewer resources than the RV-CAP controller, but in return, the AXI HWICAP controller achieves only a reconfiguration throughput of about 2% of the theoretical ICAP maximum throughput at 100 MHz (400 MB/s) [132]. During the evaluation, the RP size is defined to be 4320 LUTs, 8640 FFs, 24 DSP blocks, and 12 BRAMs and is hosting the accelerator RMs. The corresponding partial bitstream size is ~ 0.85 MiB.

Table 5.1: Hardware resource utilization of the RV-CAP controller and Xilinx AXI-HWICAP on Xilinx XCVU9P FPGA and the maximum reconfiguration throughput at a clock frequency = 100 MHz.

DPR Controller	Modules	Resource Utilization				Throughput (MB/s)
		LUTs	FFs	BRAMs	DSPs	
RV-CAP	AXI modules + Interfaces	552	1483	0	0	398.1
	Xilinx DMA Controller	2604	4010	7	0	
AXI_HWICAP	AXI modules + Interfaces	985	1023	0	0	8.23
	AXI_HWICAP	597	1364	2	0	

5.3.2 Reconfiguration Time

The reconfiguration time is measured by the performance counter register (PCCR) of the RISC-V. The reconfiguration time is measured from the API software modules including software and hardware overheads required for successfully loading a single partial bitstream file from the DDR to the FPGA configuration memory including control overheads. The measurement starts with the beginning of the data transmission of a single partial bitstream file to the DPR controller (either the RV-CAP or the AXI HWICAP) from the external DDR, and ends until the selected partial bitstream is completely transferred to the FPGA configuration memory. The total reconfiguration time is measured based on the following equation:

$$T_{tot} = T_{dec} + T_{recon} \tag{5.1}$$

where T_{dec} is the decision time required by the RISC-V core to select the RM module and configure the DPR controller, T_{recon} is the required time for the RM partial bitstream to be loaded from the external DDR and completely transferred to the FPGA configuration memory. Accordingly, for the RV-CAP and AXI HWICAP the T_{dec} = 18 µs which is a negligible time compared to the T_{recon}. Therefore, $T_{tot} \approx T_{recon}$. Figure 5.4 shows the total reconfiguration time for four RP sizes using the RV-CAP controller. The timing results using the RV-CAP controller are T_{dec} = 18 µs, T_{recon} = 2120 µs for an RP size of 0.85 MiB. As a result, the achieved reconfiguration throughput is 398.1 MB/s. Similiarly, Figure 5.5 shows the total reconfiguration time for four RP sizes using the AXI HWICAP controller. The timing results using the AXI HWICAP controller are T_{dec} = 18 µs, T_{recon} = 103175 µs for an RP size of 0.85 MiB. As a result, the achieved reconfiguration throughput is 8.23 MB/s.

5.3.3 Use Cases Accelerators

In this subsection, three basic image processing filters from the open-source HiFlipVX library [179] are used as reconfigurable hardware modules to evaluate the proposed runtime reconfiguration management with the RV-CAP controller. The three hardware accelerators

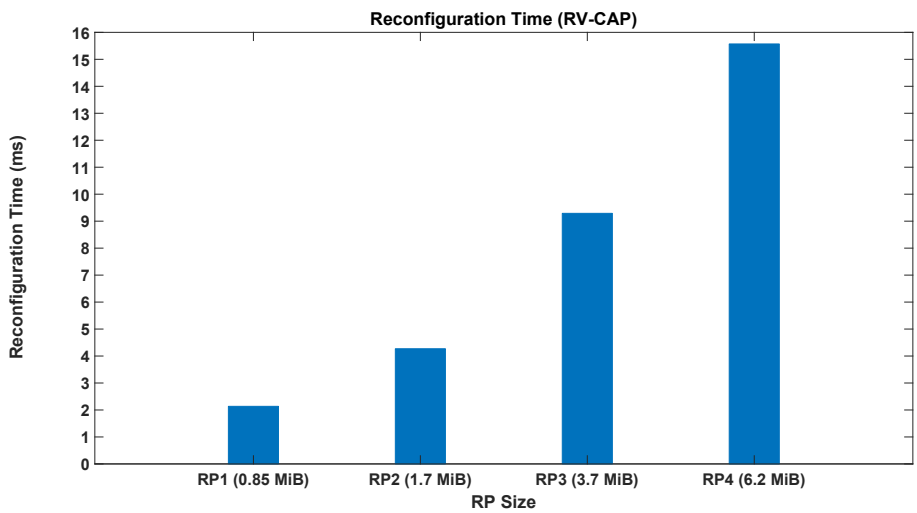


Figure 5.4: Reconfiguration time with respect to different RP sizes by using the RV-CAP controller.

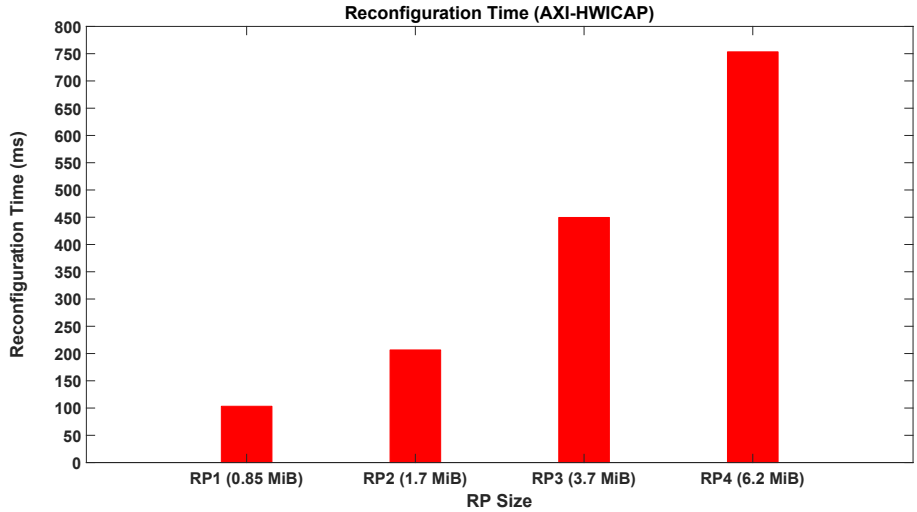


Figure 5.5: Reconfiguration time with respect to different RP sizes by using the Xilinx AXI-HWICAP [177] controller.

are hosted by a single RP, where each hardware accelerator is a RM as shown in Figure 5.1. The three filters are Sobel, Median, and Gaussian filters processing an image size of (512x512) 8-bit pixels with 256 gray values. The filters are developed using Xilinx Vivado high-level synthesis (HLS) tool with a 32-bit AXI-stream interface to be compatible with the AXIS data width of the RV-CAP controller.

The three filters are generated and synthesized separately as three RMs that are hosted by a single RP. The AXI-stream interface connects the image filter RMs and the AXI-stream R/W RM interface of the DMA inside the RV-CAP controller (as shown previously in Figure 5.2). The RMs are operating at a clock frequency of 100 MHz. The hardware accelerators do not have any connection to the SoC AXI interconnect. They are tightly coupled to the RV-CAP controller through the 32-bit AXIS interface. The image input is stored in the DDR memory to be loaded by the RV-CAP controller (in accelerator mode) after the reconfiguration process is conducted.

Figure 5.6 shows the complete floorplan of the full RISC-V-based SoC with one RP hosting the three hardware accelerator RMs. Moreover, the full RISC-V-based SoC hardware resource utilization is listed in Table 5.2 including the RP size and the utilization of each RM from the RP total available hardware resources. Table 5.2 shows that the RV-CAP controller consumes $\sim$ 3.9%, $\sim$ 7.3% of the total RISC-V-based SoC hardware resources in terms of LUTs and FFs respectively. Therefore, the RV-CAP controller has a very small resource utilization overhead which makes it suitable to be integrated for small and low-power FPGA devices as well.

Furthermore, Table 5.3 shows the total execution time for the three image filter RMs including reconfiguration and computing time. As the total execution time is computed as follows:

$$T_{tot} = T_{dec} + T_{recon} + T_{comp} \tag{5.2}$$

Where T_{dec} is the decision time required by the RISC-V core to select the RM module and configure the DPR controller, T_{recon} is the required time for the RM partial bitstream to be loaded from the external DDR and completely transferred to the FPGA configuration memory, and T_{comp} is the accelerator computation time including input and output data transfer to the DDR memory to filter a single image and write back the output to the DDR memory. As a result

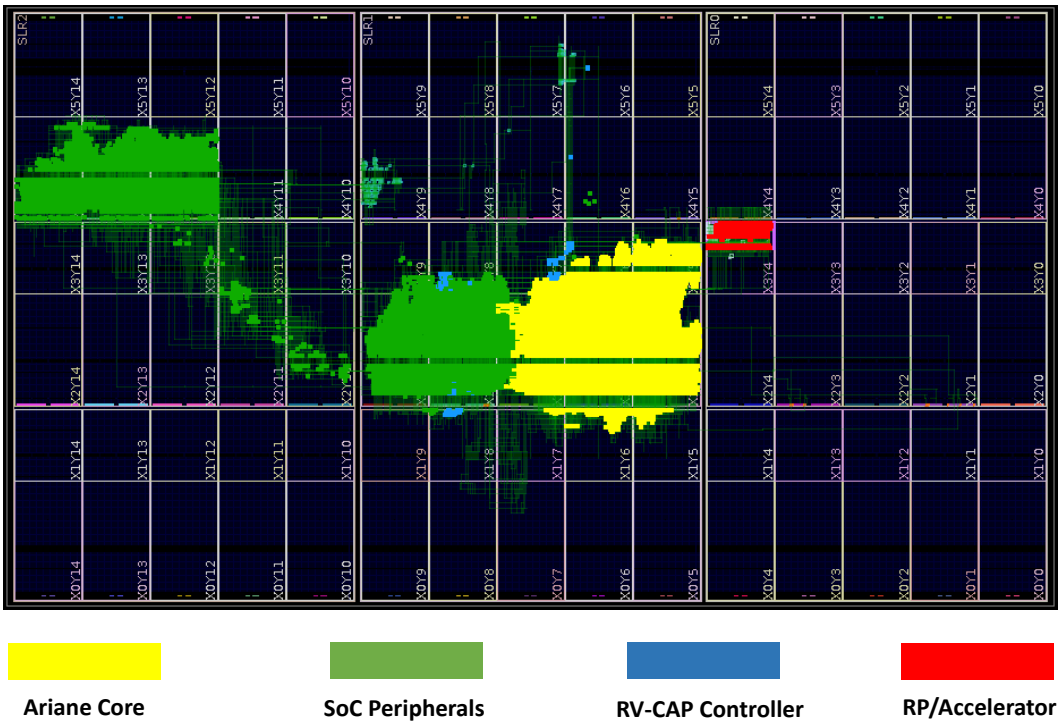


Figure 5.6: An overview of the self-adaptive RISC-V based SoC floorplan on a Xilinx Virtex Ultrascale+ (XCVU9P) FPGA.

(from Table 5.3), the reconfiguration time T_{recon} is ~2.5X the computation time as the used accelerated functions execute few operations and not a compute-intensive kernel. Therefore, in case of medium-workload applications, the reconfiguration time would be negligible to the computation time. As a result, the proposed runtime reconfiguration management with the RV-CAP controller is suitable for high-speed self-adaptive RISC-V SoC.

Table 5.2: Hardware resource utilization of the self-adaptive RISC-V based SoC with a single RP to host multiple image processing accelerator modules on Xilinx XCVU9P FPGA.

SoC Components		Resource Utilization			
		LUTs	FFs	BRAMs	DSPs
RV64IMAC Core		39781	22489	36	27
Peripherals + Boot Memory		32727	38023	32	0
RV-CAP		3156	5493	7	0
Reconfigurable Partition (RP)		4320	8640	12	24
Full SoC		79984	74645	87	51
Reconfigurable Modules (RMs)	Gaussian Filter	901	773	4	0
	Median Filter	2325	998	2	0
	Sobel Filter	1830	3224	2	16

Table 5.3: Image processing accelerators execution and reconfiguration time at a clock frequency = 100 MHz.

HW Accelerator	Decision Time (T_{dec}) (μs)	Reconfiguration Time (T_{recon}) (μs)	Compute Time (T_{comp}) (μs)	Total Time (T_{tot}) (μs)
Gaussian Filter	18	2120	606	2744
Median Filter	18	2120	598	2736
Sobel Filter	18	2120	588	2726

5.4 Reconfiguration Management Unit Integration into the Tile-based Many-Core Architecture

The proposed multi-ISA tile-based many-core architecture differentiates from other state-of-the- art many-core architectures by the ability to support different many-core configurations throughout the runtime adaptation feature presented in previous sections. This section presents and evaluates the integration of the runtime reconfiguration management discussed before with the multi-ISA tile-based many-core architecture presented in Chapter 3.

The proposed many-core architecture consists of a static partition region and several RPs to be reconfigured according to the selected many-core configuration. The static region hosts the main processing tile and the NoC architecture. While RPs host different configurations for 64-/32-bit compute tiles (e.g. number of PEs, memory type and size) through a set of compute tiles RMs. All compute tiles RMs share unified interfaces to the NoC routers through the NI with single domain clock and reset signals. The DPR process is conducted internally through the ICAP primitive to allow the tile-based many-core architecture to self-manage the configuration process without any external controlling peripherals (e.g. a PC through a JTAG). The aforementioned reconfiguration management approach with the RV-CAP controller is integrated inside the main processing tile in a similar way to integration with a RISC-V-based SoC.

Figure 5.7 shows a schematic overview of the main processing tile with the RV-CAP controller attached to the AXI interconnect. The main processing tile is using a 64-bit AXI-4 interconnect where four RV64IMAC cores are connected with shared instruction memory, a NI, and external peripherals. All four RISC-V core are master units on the AXI-4 interconnect, while the rest of the peripherals are slave memory-mapped components within the core's address space ranges including the RV-CAP controller. The reconfiguration management APIs software module can be executed on any RISC-V core based on a predefined task mapping by the user. The main processing tile is running on a clock frequency of 120 MHz while the RV-CAP is running at 100 MHz. Therefore, an extra AXI clock converter is inserted for the two clock domains.

Moreover, a slight modification to the RV-CAP controller is conducted by removing the AXIS switch and the configuration mode interface to support only the DPR mode of operation. In the case of integration to the main processing tile, no extra reconfigurable hardware

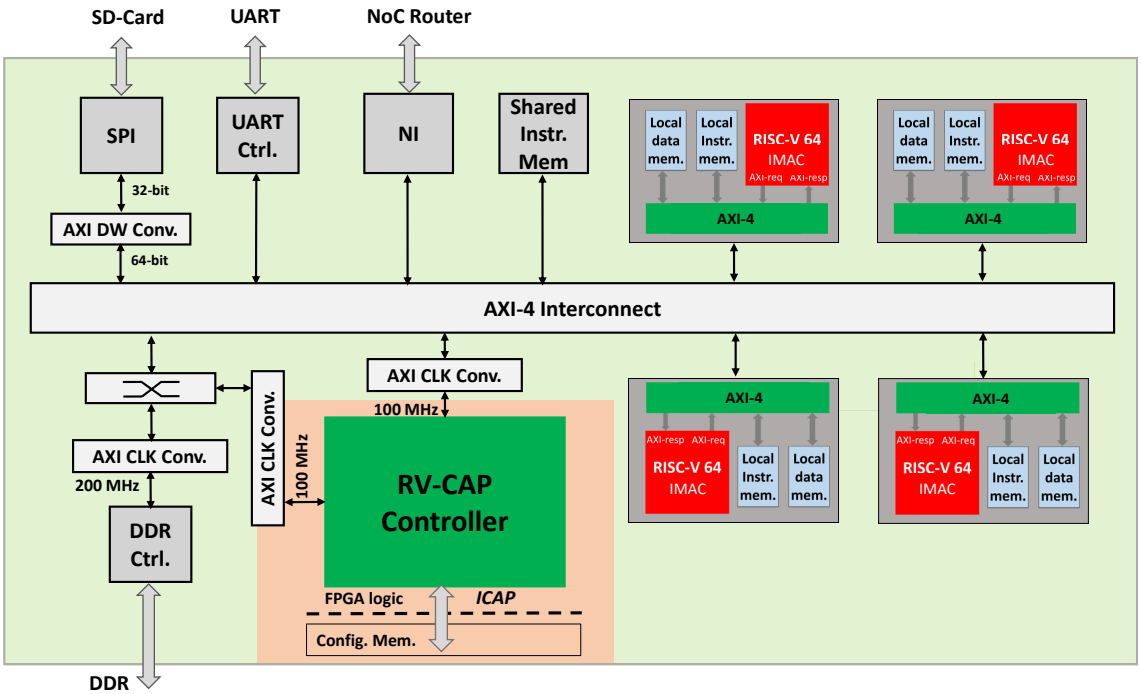


Figure 5.7: A schematic overview of the main processing tile with the RV-CAP controller.

accelerators are attached to the AXI interconnect. Therefore the RV-CAP does not require the extra AXI modules for operation mode selection. A block diagram of the modified RV-CAP controller is shown in Figure 5.8. The proposed implementation provides a high data throughput rate to the FPGA configuration memory via the ICAP primitive. The RV-CAP controller has only three interfaces as shown in Figure 5.8 one control interface connected to the main tile AXI interconnect to control the Xilinx DMA controller, a data interface for transferring partial bitstreams from the external DDR to the ICAP primitive, and the ICAP primitive interface to the FPGA configuration memory.

The DMA controller is connected to the main processing tile DDR controller through an additional crossbar as a master component to the DDR controller. Therefore, the DDR can be accessed either by RISC-V PEs as a shared data memory or by the RV-CAP controller to load a partial bitstream to the FPGA configuration memory. Hence, partial bitstreams are stored in different address ranges than shared data memory address section of RISC-V PEs. As the RV-CAP controller is implemented inside the 64-bit main processing tile, the DMA is configured to transfer a 64-bit data word from the DDR. Therefore, as mentioned in Figure 5.8, an AXIS2ICAP block is implemented to split a 64-bit data word into 2x32-bit data words to be compatible with the 32-bit data interface of the ICAP primitive. Besides, the valid stream signal is inverted and connected to the ICAP data port as the write enable signal (ICAP_WE). Also, an AXI data width and protocol converters are used to convert between the 64-bit AXI- interconnect and the 32-bit AXI-Lite control interface of Xilinx DMA component. All reconfiguration steps mentioned in Section 5.2 are managed by one 64-bit PE through the same set of RV-CAP software modules as shown in Listing 5.1, 5.2, 5.3.

For evaluation, two NoC 2-D mesh size configurations are used to support several many-core sizes regarding the number and types of compute tiles. As mentioned previously, the main processing tile and NoC are hosted by the static partition region of the architecture, while

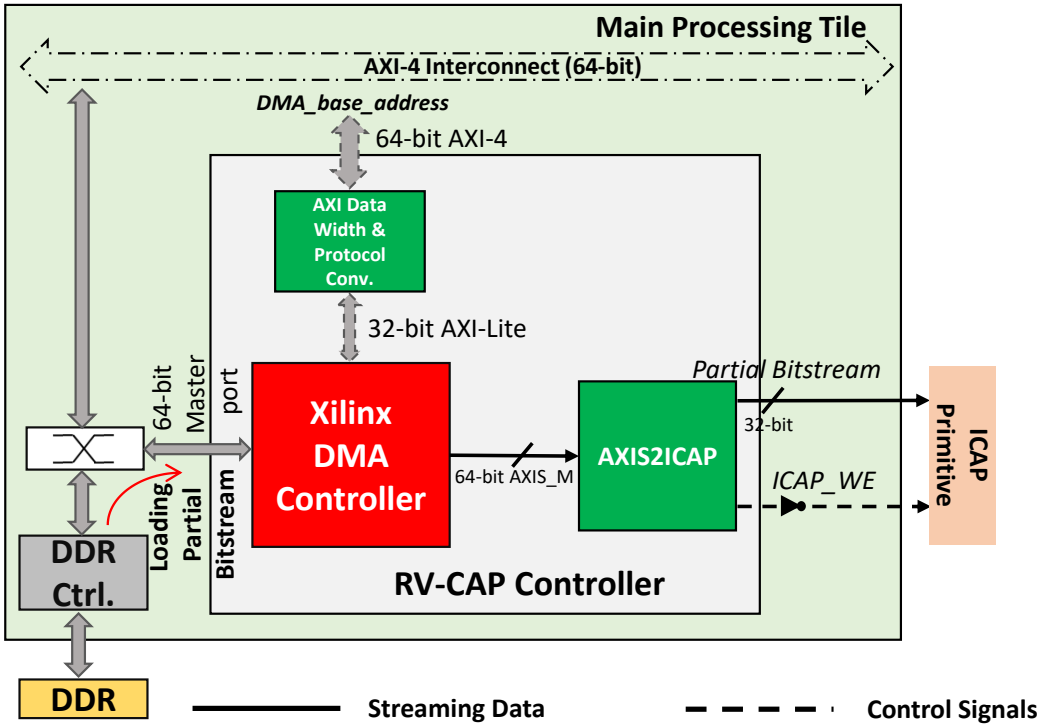


Figure 5.8: A detailed block diagram of the RV-CAP controller within the main processing tile for adaptive tile-based many-core architecture.

the other three types of compute tiles are swapped over multiple RPs at run-time based on the tile-based many-core size. Figure 5.9 and Figure 5.10 show two FPGA floorplans based on two different NoC configurations as follows:

1. The first tile-based many-core configuration has a 2x7 NoC size with 12xRPs (same size) to host multiple 32-/64-bit compute tile modules with a static main processing tile.
2. The second tile-based many-core configuration has a 2x4 NoC size with 7xRPs (same size) to host multiple 32-/64-bit compute tile modules with a static main processing tile.

The first tile-based many-core size is shown in Figure 5.9, it consists of 12 RPs that can host two types of compute tiles (32-bit, and 64-bit (w/single-PE) tiles) as RMs. All 12 RPs have the same size on the FPGA floorplan with the same number and types of hardware resources. A single RP size is specified to host the largest compute tile used in the first tile-based many-core size.

Table 5.4 shows hardware resource utilization for a single RP and percentage resource utilization for each RM from the total RP size. The maximum utilization percentage is achieved by the 64-bit (w/single-PE) compute tile with (87.5%) LUTs and (91.6%) on-chip memory utilization (BRAM+URAM). On the other hand, the second tile-based many-core size with 2x4 NoC supports larger RP size to support the largest configurable compute tile module (64-bit (w/2-PE) tile) for the tile-based many-core architecture. Figure 5.10 shows the FPGA floorplan of the second tile-based many-core size with 7 RPs and three types of RMs for all compute tiles. However, the number of compute tiles is limited to 7 compute tiles to fit with the target FPGA total size. Similar to the first many-core, all RPs have the same size that can fit with the largest compute tile which is in this case the 64-bit compute tile with dual PEs.

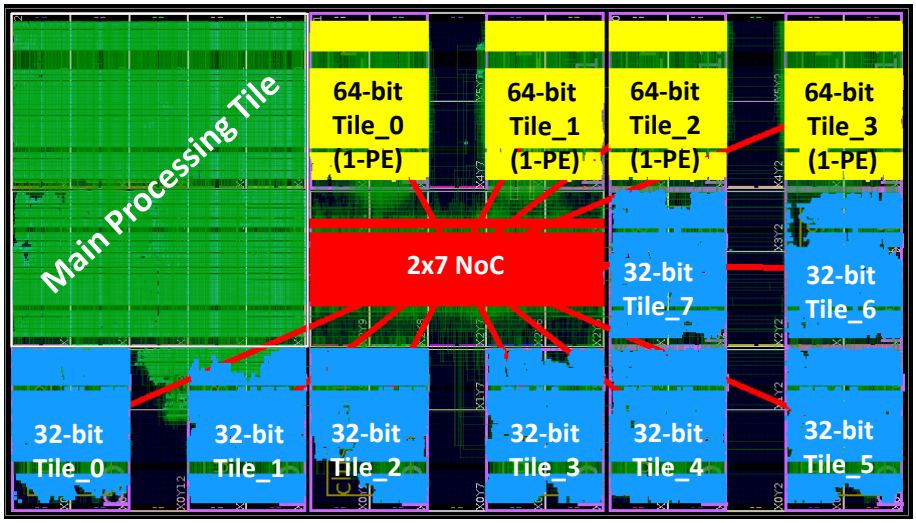


Figure 5.9: FPGA floorplan of the first tile-based many-core size with 2x7 NoC configured by 8x32-bit (w/4-PEs), and 4x64-bit (w/single-PE) compute tiles (12xRPs).

Table 5.4 shows resource utilization for the second tile-based many-core configuration and its RMs. The RP size is double the size used for the first many-core configuration with maximum utilization of (92.2%) for LUTs and (87.5%) for on-chip memory hosting 64-bit (w/2-PEs) compute tile modules. In contrast, 32-bit compute tile modules consume less than (30%) of the total RP resources. Therefore, the second tile-based many-core configuration can be used efficiently to support more 64-bit compute tiles in comparison with the first many-core configuration. Moreover, Table 5.5 shows the total resource utilization for the two many-core configurations shown in Figure 5.9 and Figure 5.10. The resource utilization reports the total hardware resources required by all used RPs with the main processing tile for the two many-core configurations.

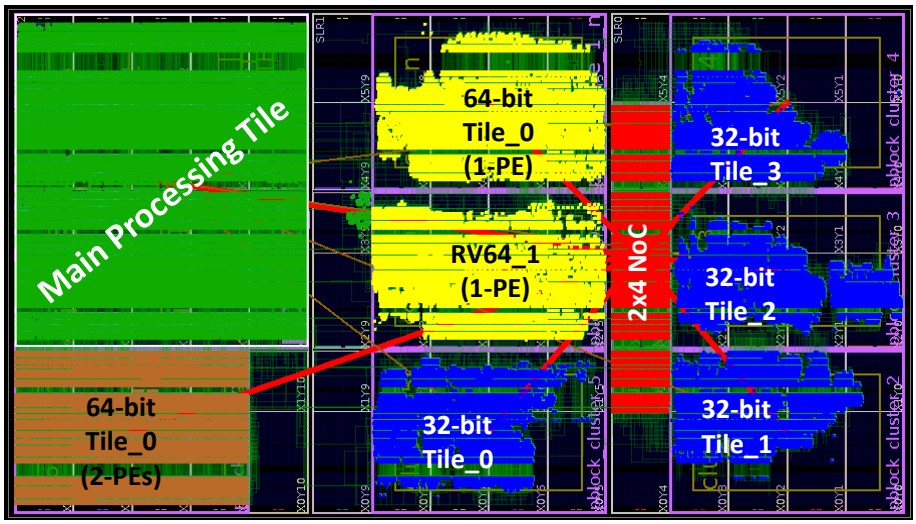


Figure 5.10: FPGA floorplan of the second tile-based many-core size with 2x4 NoC configured by 4x32-bit (w/4-PEs), 2x64-bit (w/single-PE), and 1x64-bit (w/2-PEs) compute tiles (7xRPs).

Table 5.4: DPR resource utilization and reconfiguration time for two tile-based many-core configurations on Xilinx XCVU9P FPGA.

Many-Core Configuration (#RPs)			Resource Utilization					Reconfiguration Time
			LUTs	FFs	BRAMs	URAMs	DSPs	
First Many-Core Configuration (12 RPs)	Single RP Size (Single Tile)		52880	105760	96	32	384	18.8 ms
	RMs	32-bit-Tile (4-PEs)	30717 (58%)	15359 (14.5%)	35 (36.5%)	12 (37.5%)	24 (6.25%)	
		64-bit-Tile (Single-PE)	46311 (87.5%)	23156 (21.9%)	88 (91.6%)	8 (0.25%)	27 (7%)	
Second Many-Core Configuration (7 RPs)	Single RP Size (Single Tile)		103680	207360	192	64	768	38.1 ms
	RMs	32-bit-Tile (4-PEs)	30717 (29.6%)	15359 (7.4%)	35 (18.2%)	12 (18.7%)	24 (3.1%)	
		64-bit-Tile (Single PE)	46311 (44.6%)	23156 (11.2%)	88 (45.8%)	8 (12.5%)	27 (3.5%)	
		64-bit-Tile (2-PEs)	95636 (92.2%)	47818 (23.1%)	168 (87.5%)	8 (12.5%)	54 (7%)	

Table 5.5: Total hardware resource utilization for the two different many-core sizes shown in Figure 5.9, Figure 5.10 on Xilinx XCVU9P FPGA.

Many-Core configuration	Resource Utilization				
	LUTs	FFs	BRAMs	URAMs	DSPs
First Many-Core Configuration (2x7 NoC, 12 RPs)	853333 (72.1%)	426997 (18.1%)	1500 (69.5%)	384 (40%)	4722 (69%)
Second Many-Core Configuration (2x4 NoC, 7 RPs)	944533 (80%)	521258 (22.2%)	1692 (78.3%)	448 (46.7%)	5490 (80.3%)

Accordingly, the first many-core configuration is more suitable for heterogeneous configurations that support more 32-bit compute tiles than 64-bit compute tiles to increase the efficiency of resources usability. On the other hand, the second many-core configuration is more suitable for 64-bit based many-core configurations. In addition to that, several many-core configurations can be realized by using different NoC sizes to support a different number of homogeneous or heterogeneous compute tiles based on the target application requirements.

Furthermore, reconfiguration time is an important aspect to evaluate the performance of the internal reconfiguration manager/RV-CAP within the tile-based many-core architecture. Therefore, the reconfiguration time is measured through the RISC-V core PCCR to measure the required number of clock cycles from loading a partial bitstream from the DDR memory until fully transferring the partial bitstream to the ICAP primitive. The reconfiguration time includes all software and hardware overhead required by the reconfiguration manager/RV-CAP to successfully loaded one partial bitstream to the FPGA configuration memory through the ICAP.

As shown in Table 5.4 (last column), the total reconfiguration time of a single RP for the first many-core configuration is 18.8 ms. For the second many-core configuration, as the RP size increased, the required reconfiguration time is 38.1 ms. Accordingly, the proposed reconfiguration manager supports a high-speed reconfiguration process as it uses a separate data stream channel to transfer partial bitstream to the ICAP primitive through a Xilinx DMA.

5.5 Summary

Chapter 5 presents a novel runtime reconfiguration management approach for self-adaptive FPGA-based RISC-V SoC. The reconfiguration management approach is suitable for single-, muli, and many-core RISC-V-based architectures. The reconfiguration management approach has a novel DPR controller called RV-CAP to enable DPR on FPGA-based RISC-V SoCs. The RV-CAP supports the management of DPR within a programmable software environment through a set of developed software drivers running on a RISC-V core. Moreover, the RV-CAP is considered a high-speed DPR controller that allows a high reconfiguration throughput that reaches 398.1 MB/s. In addition, the RV-CAP controller supports the managing of a single RP and its hosted RMs by providing controlled stream interfaces between the SoC external DDR memory and the corresponding RM hardware accelerators hosted by a single

RP. The runtime reconfiguration management is integrated into the main processing tile of the developed tile-based many-core architecture proposed by this dissertation to allow the run-time configuration to change numbers and types of compute tiles for several tile-based many-core configurations. Hence, the integration of the runtime reconfiguration management with the tile-based many-core architecture provides the self-adaptation feature to realize several heterogeneous tile-based many-core configurations and taxonomies at run-time.

Section 5.1 presents the internal DPR management for self-adaptive RISC-V SoC. It describes and presents the used open-source 64-bit RISC-V-based SoC and its internal components. Moreover, the required modifications to the open-source SoC to deploy the proposed reconfiguration management approach from the hardware and software sides. A set of software drivers to access the SoC I/O peripherals have been developed in order to load the partial bitstreams from an external SD-card to the SoC DDR memory. In addition, a set of utility modules to communicate with memory-mapped peripherals for reading and writing data through the core address space range.

From the hardware side, the RV-CAP controller is developed and implemented as a memory-mapped peripheral to the RISC-V core. The RV-CAP controller supports two modes of operation: the DPR mode, and the accelerator mode. In the DPR mode of operation, the RV-CAP is used to manage partial bit stream data transfer between the external DDR and the internal configuration memory. On the other hand, for the accelerator mode of operation, the RV-CAP is used to manage stream data transfer from/to hardware accelerator modules hosted by a single RP and the external DDR memory. The RV-CAP is a master unit when it interacts with the DDR controller through the DMA for read and write stream data. The RV-CAP is working at a single clock source of 100 MHz in a fully synchronized design.

Section 5.2 presents the software side of the reconfiguration management approach including the developed set of APIs and software modules to control and manage the reconfiguration process from a software programmable environment on a RISC-V core. This section presents and describes the RV-CAP APIs to manage and abstract the reconfiguration process from the RISC-V core as well as the potential to use a DPR vendor controller and control it from the RISC-V core. The interaction between the RV-CAP unit and RISC-V core is conducted through a set of API software modules. Where the RISC-V core manages the reconfiguration process by selecting the desired RMs to be loaded based on application requirements using the three following controlling steps. The first step is initializing all RMs by reading all partial bitstream files from the SD-Card and storing them on the external DDR memory. The second and third steps are the selection of mode configuration to be the DPR mode of operation and afterwards start the reconfiguration process. Furthermore, an alternative DPR controller unit (AXI HWICAP) is used to allow the RISC-V core to take over the task of the master unit to manage the writing process to the FPGA configuration memory through the ICAP primitive. Also, some modifications have been done to integrate the AXI HWICAP IP core into the RISC-V-based SoC.

Section 5.3 presents the evaluation and obtained experimental results of the proposed runtime reconfiguration management. The runtime reconfiguration management is evaluated based on the hardware resource utilization of the RV-CAP controller, and the reconfiguration throughput using different RPs sizes and image processing based hardware accelerators as RMs. The RV-CAP features a low resource utilization overhead which makes it suitable for small and low-power FPGAs as well as large FPGA devices. The reconfiguration time is measured from the API software modules including software and hardware overheads

required for successfully loading a single partial bitstream file from the DDR to the FPGA configuration memory including control overheads. The RV-CAP controller achieves a reconfiguration throughput of 398.1 MB/s. On the other hand, by using the AXI HWICAP controller a reconfiguration throughput of 8.23 MB/s can be achieved. Furthermore, three basic image processing hardware accelerated filters are used as reconfigurable hardware modules to evaluate the proposed runtime reconfiguration management with the RV-CAP controller. It shows the feasibility of the reconfiguration management approach to be used for high-speed self-adaptive RISC-V-based SoC.

Finally, Section 5.4 presents and evaluates the integration of the reconfiguration management unit (RV-CAP) into the main processing tile of the target tile-based many-core architecture proposed by this dissertation. The tile-based many-core architecture consists of a static partition region and several RPs to be reconfigured according to the selected many-core configuration. The static region hosts the main processing tile and the NoC architecture. While RPs host different configurations for 64-/32-bit compute tiles. The DPR process is conducted internally through the ICAP primitive to allow the tile-based many-core architecture to self-manage the configuration process without any external controlling peripherals. For evaluation, two tile-based many-core configurations with different numbers and sizes of RPs are used to support several heterogeneous many-core taxonomies regarding the number and types of compute tiles.

6 Conclusion and Outlook

This dissertation entails three major contributions that together provide an agile platform for the design and realization of heterogeneous tile-based many-core architectures. In this concluding chapter, the main challenges of this research work are re-iterated with a summary of the major contributions. In the last section of this chapter, future work and extensions based on the proposed contributions are discussed and highlighted.

6.1 Summary of Contributions

Current and future multi- and many-core SoC architectures are characterized by their growing number of heterogeneous computing elements due to the unprecedented computing requirements posed by new trends in artificial intelligence and signal processing applications. The main key challenge arises from system-level integration issues and the huge amount of design complexities related to the integration and interaction between heterogeneous compute units. Moreover, design specifications could vary due to different application requirements which leads to the necessity of a new design process for each new application requirement. This leads to a limited degree of design extensibility and increases the cost of post-design upgrades. Therefore, a modular and reusable (agile) platform for many-core architectures realization is proposed by this dissertation in order to provide a unified system-level architecture for heterogenous and homogeneous many-core systems. The proposed agile platform enables the development of scalable many-core architectures using modular and reusable heterogeneous compute tiles. The many-core platform is realized on top of a 2-D mesh NoC-based architecture where a unified communication and programming method is used over heterogeneous compute tiles. In addition, the many-core platform features a high-speed internal reconfiguration management approach for self-adaptation to multiple many-core configurations at runtime. In this dissertation, the many-core platform is targeting FPGA devices where many-core architectures are realized as FPGA-based overlays.

The new proposed many-core platform is developed and realized based on three main contributions: (1) supporting heterogeneous ISAs through modular tile-based many-core architecture, (2) seamless integration of custom hardware accelerators, and (3) many-core self-adaptation management. The proposed contributions are summarized as follows:

Modular Tile-based Many-Core Architecture for Heterogeneous ISAs

Heterogeneous compute tiles are the core of the proposed platform, they represent the computing nodes for realized many-core systems. The tile-based platform supports three types of heterogeneous tiles that support multiple RISC-V ISAs in addition to custom hardware accelerators. Two RISC-V based compute tiles based on different 32-/64-bit RISC-V ISAs are realized. Each tile consists of single or multi-RISC-V-based PEs. RISC-V-based PE hosts a single core RISC-V with tightly coupled scratchpad memories for instruction and data to serve as local PE memory to increase data locality. All RISC-V-based PEs are equipped with AXI-based interfaces to support seamless integration into RISC-V based compute tiles. The 32-bit tile supports RV32IMC ISA based on an open-source RV32 core, and the 64-bit tiles support RV64IMAC based on an open-source RV64 core. All compute tiles have a regular design pattern based on a bus-based architecture with shared data/instruction memory and a memory-mapped network interface with the option to be augmented with optional loosely coupled hardware accelerators. The third general-purpose tile is a permanent processing tile in the proposed platform called the main processing tile. The main processing tile controls and manages external many-core peripherals (i.e. SD-card, UART) and it can be extended to support other types of off-chip peripherals. Furthermore, the reconfiguration management unit and its associated components are hosted and managed by the main processing tile. Also, a message-passing communication model is developed for tile-to-tile communication over the NoC as well as a unified parallel programming method over different general-purpose tiles.

Hybrid Memory/Accelerator Tile Architecture for Tile-based Many-Core Systems

In order to increase the level of heterogeneity to support the integration of custom hardware accelerators, a novel hybrid memory/accelerator tile architecture is developed. The hybrid tile is a modular and reusable tile that can be configured at run-time to operate as a scratchpad shared memory between compute tiles or as an accelerator tile hosting a local hardware accelerator logic. The hybrid tile is designed and implemented to be seamlessly integrated into the proposed tile-based platform through the NoC. It consists of two main architectural units: a control unit that receives control packets and configures the tile based on the mode of operation, and a data path that includes memory read, write managers, hardware accelerator wrapper connected to on-chip memory modules. The interaction between general-purpose tiles and the hybrid tile is conducted through a message-based protocol. Where general-purpose tiles request to access the on-chip memory or the accelerator through a set of control messages. The control unit handles all memory and accelerator requests from all general-purpose tiles to configure the data-path based on the requested mode of operation and associated parameters. The tile data path is responsible to establish data paths between tile's NI and on-chip memory based on control signals from the control unit. Three categories of data paths can be established: memory read path, memory write path, and accelerator path. Memory read and write paths can be established in parallel to handle read and write memory requests simultaneously. On the other hand, only the accelerator path can be solely established as the hardware accelerator logic requires to use on-chip memory as local memory for load/store operations.

Reconfiguration Management for Self-Adaptive Tile-based Systems

A reconfiguration management approach is proposed to allow self-adaptation for the proposed tile-based platform. The applied self-adaptation method is based on self-controlling

and management of the reconfiguration process through the main processing tile. The internal reconfiguration process relies on a novel DPR controller targeting FPGA design flow for RISC-V-based SoC (RV-CAP) to change the types and functionalities of compute tiles at run-time. The DPR process is conducted internally through the internal access configuration port (ICAP) primitive. The tile-based platform consists of a static partition region and several reconfigurable partition (RP) regions to be reconfigured according to the selected many-core configuration. The static region hosts the main processing tile and the NoC architecture. While the reconfigurable regions host different configurations for 64-/32-bit compute tiles and hybrid tiles through a set of tiles reconfigurable modules (RMs). All tiles RMs share unified interfaces to the NoC routers through NI with single domain clock and reset signals.

This dissertation proposes an agile tile-based platform for heterogeneous many-core systems. The platform is based on a scalable and modular tile-based architecture that features a high degree of heterogeneity supporting heterogeneous ISAs as well as custom hardware accelerators for general and domain-specific workloads. Also, the proposed platform supports self-adaptation to realize several many-core configurations and taxonomies at run-time. Moreover, it aims to ease the development and realization of heterogeneous many-core architectures by reducing the design time and the non-recurrent engineering costs.

6.2 Future Work

Future research directions and extensions of this work are various. This dissertation focused on the design and development of an agile tile-based many-core platform targeting only FPGA devices. This dissertation lays the foundation for several research work on heterogeneous computing systems for both high-performance and near-threshold computation. The following are some potential future developments.

Chiplet-based heterogeneous system

Chiplet designs are currently used for exascale computing systems which require more computing resources and performance. The current version of the proposed tile-based many-core platform targets only FPGA devices, specifically a single FPGA device as discussed in Chapter 3 and Chapter 5. For future improvements, different many-core configurations and taxonomies can be realized on multiple FPGA devices equipped with high-performance communication fabric between them. Due to the limited maximum clock frequency achieved by FPGAs, bringing the tile-based many-core platform to silicon in the form of a custom ASIC design would improve the computing performance and energy efficiency. Therefore, increasing the scalability and heterogeneity of the tile-based platform can be achieved by exploring a chiplet-based design to combine multiple heterogeneous many-core instants together. Moreover, chiplet-based design could allow the integration of various kinds of new memory technology (e.g. HBM, HMC) to the many-core platform in order to increase the memory bandwidth for memory-intensive applications.

Near-threshold computing

High energy efficiency can be achieved by exploring near-threshold techniques with high operating frequencies. Near-threshold computing can be applied to digital SoC for more

energy efficiency by reducing transistors supplying voltage to threshold voltage on the circuit level, where leakage and dynamic energy achieve the minimum point. The current version of the tile-based many-core platform does not support power management units and voltage scaling controllers in order to manage and control the supply voltage to the processing cores or the hardware accelerators. In order to explore power management and near-threshold computing, the proposed tile-based platform has to be realized as an ASIC design first. The FPGA-based version has a limitation to control the supply voltage on a specific partition on the FPGA floorplan. For example, controlling the supply voltage of a specific compute tile is not possible using the current FPGA technology. Therefore, a custom ASIC design will provide more freedom to control the supply voltage of a specific PE, compute tile, or a custom hardware accelerator.

Bibliography

[1] Inyup Kang. "The Art of Scaling: Distributed and Connected to Sustain the Golden Age of Computation". In: *2022 IEEE International Solid- State Circuits Conference (ISSCC)*. Vol. 65. 2022, pp. 25–31. DOI: 10.1109/ISSCC42614.2022.9731536.

[2] John L Hennessy and David A Patterson. "A new golden age for computer architecture". In: *Communications of the ACM* 62.2 (2019), pp. 48–60.

[3] Ashish Nayak, HsinChen Chen, Hugh Mair, Rolf Lagerquist, Tao Chen, Anand Rajagopalan, Gordon Gammie, Ramu Madhavaram, Madhur Jagota, CJ Chung, Jenny Wiedemeier, Bala Meera, Chao-Yang Yeh, Maverick Lin, Curtis Lin, Vincent Lin, Jiun Lin, YS Chen, Barry Chen, Cheng-Yuh Wu, Ryan ChangChien, Ray Tzeng, Kelvin Yang, Achuta Thippana, Ericbill Wang, and SA Hwang. "A 5nm 3.4GHz Tri-Gear ARMv9 CPU Subsystem in a Fully Integrated 5G Flagship Mobile SoC". In: *2022 IEEE International Solid- State Circuits Conference (ISSCC)*. Vol. 65. 2022, pp. 50–52. DOI: 10.1109/ISSCC42614.2022.9731604.

[4] Jonathan Balkind, Katie Lim, Michael Schaffner, Fei Gao, Grigory Chirkov, Ang Li, Alexey Lavrov, Tri M. Nguyen, Yaosheng Fu, Florian Zaruba, Kunal Gulati, Luca Benini, and David Wentzlaff. "BYOC: A "Bring Your Own Core" Framework for Heterogeneous-ISA Research". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 699–714. DOI: 10.1145/3373376.3378479.

[5] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs". In: *IEEE Micro* 40.4 (2020), pp. 10–21. DOI: 10.1109/MM.2020.2996616.

[6] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. "The risc-v instruction set manual, volume i: Base user-level isa". In: *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62* 116 (2011).

[7] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. "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. DOI: 10.1109/ISCA.2018.00014.

[8] Davide Giri, Kuan-Lin Chiu, Giuseppe Di Guglielmo, Paolo Mantovani, and Luca P. Carloni. "ESP4ML: Platform-Based Design of Systems-on-Chip for Embedded Machine Learning". In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2020, pp. 1049–1054. DOI: 10.23919/DATE48585.2020.9116317.

[9] Luca P. Carloni. "Scalable Open-Source System-on-Chip Design: (Invited Talk - Extended Abstract)". In: *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*. 2020, pp. 7–9. DOI: 10.1109/VLSI-SOC46417.2020.9344077.

[10] Sven Rheindt, Temur Sabirov, Oliver Lenke, Thomas Wild, and Andreas Herkersdorf. "X-Centric: A Survey on Compute-, Memory- and Application-Centric Computer Architectures". In: *The International Symposium on Memory Systems*. Association for Computing Machinery, 2020, pp. 178–193. DOI: 10.1145/3422575.3422792.

[11] Salma Hesham, Diana Göhringer, and Mohamed Abd El Ghany. "ARTNoCs: An Evaluation Framework for Hardware Architectures of Real-Time NoCs". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, pp. 259–264. DOI: 10.1109/IPDPSW.2016.87.

[12] MV Arunkumar and GH Hayatnagarkar. "RISKA: Towards an open-source RISC-V based domain-specific system-on-chip for SKA data processing". In: *Proc. Workshop Comput. Archit. Res. With RISC-V (CARRV) Workshop (ISCA)*. 2021, pp. 1–7.

[13] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. "Agile SoC Development with Open ESP". In: *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD '20)*. New York, NY, USA: Association for Computing Machinery, 2020. DOI: 10.1145/3400302.3415753.

[14] John Shalf. "The future of computing beyond Moore's Law". In: *Philosophical Transactions of the Royal Society A* 378.2166 (2020), p. 20190061.

[15] Ahmed Kamaleldin, Salma Hesham, and Diana Göhringer. "Towards a Modular RISC-V Based Many-Core Architecture for FPGA Accelerators". In: *IEEE Access* 8 (2020), pp. 148812–148826. DOI: 10.1109/ACCESS.2020.3015706.

[16] Ahmed Kamaleldin and Diana Göhringer. "AGILER: An Adaptive Heterogeneous Tile-Based Many-Core Architecture for RISC-V Processors". In: *IEEE Access* 10 (2022), pp. 43895–43913. DOI: 10.1109/ACCESS.2022.3168686.

[17] Ahmed Kamaleldin, Muhammad Ali, Pedram Amini Rad, Marcus Gottschalk, and Diana Göhringer. "Modular Memory System for RISC-V Based MPSoCs on Xilinx FPGAs". In: *International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, Oct. 2019, pp. 68–73. DOI: 10.1109/MCSoC.2019.00017.

[18] Ahmed Kamaleldin and Diana Göhringer. "A Hybrid Memory/Accelerator Tile Architecture for FPGA-based RISC-V Manycore Systems". In: *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, Aug. 2022, pp. 1–7. DOI: 10.1109/FPL57034.2022.00053.

[19] Ahmed Kamaleldin and Diana Göhringer. "Design For Agility: A Modular Reconfigurable Platform for Heterogeneous Many-Core Architectures". In: *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, Aug. 2021, pp. 265–266. DOI: 10.1109/FPL53798.2021.00050.

[20] Ahmed Kamaleldin and Diana Göhringer. "An Agile Tile-based Platform for Adaptive Heterogeneous Many-Core Systems". In: *2022 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, Dec. 2022, pp. 1–4. DOI: 10.1109/ICFPT56656.2022.9974358.

[21] Najdet Charaf, Ahmed Kamaleldin, Martin Thümmler, and Diana Göhringer. "RV-CAP: Enabling Dynamic Partial Reconfiguration for FPGA-Based RISC-V System-on-Chip". In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2021, pp. 172–179. DOI: 10.1109/IPDPSW52791.2021.00033.

[22] Mark Bohr. "A 30 year retrospective on Dennard's MOSFET scaling paper". In: *IEEE Solid-State Circuits Society Newsletter* 12.1 (2007), pp. 11–13.

[23] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark Silicon and the End of Multicore Scaling". In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA '11. San Jose, California, USA, 2011, pp. 365–376. DOI: 10.1145/2000064.2000108.

[24] Mark D. Hill and Michael R. Marty. "Amdahl's Law in the Multicore Era". In: *Computer* 41.7 (2008), pp. 33–38. DOI: 10.1109/MC.2008.209.

[25] Wayne Wolf, Ahmed Amine Jerraya, and Grant Martin. "Multiprocessor System-on-Chip (MPSoC) Technology". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.10 (2008), pp. 1701–1713. DOI: 10.1109/TCAD.2008.923415.

[26] Shekhar Borkar. "Thousand Core Chips: A Technology Perspective". In: *Proceedings of the 44th Annual Design Automation Conference*. Association for Computing Machinery, 2007, pp. 746–749. DOI: 10.1145/1278480.1278667.

[27] Lionel Torres, Pascal Benoit, Gilles Sassatelli, Michel Robert, Fabien Clermidy, and Diego Puschini. "An introduction to multi-core system on chip–trends and challenges". In: *Multiprocessor System-on-Chip* (2011), pp. 1–21.

[28] Peter M. Kogge and Brian A. Page. "Locality: The 3rd Wall and the Need for Innovation in Parallel Architectures". In: *Architecture of Computing Systems: 34th International Conference, ARCS 2021, Virtual Event, June 7–8, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 3–18. DOI: 10.1007/978-3-030-81682-7_1.

[29] Wm. A. Wulf and Sally A. McKee. "Hitting the Memory Wall: Implications of the Obvious". In: *SIGARCH Comput. Archit. News* 23.1 (Mar. 1995), pp. 20–24. DOI: 10.1145/216585.216588.

[30] Qiang Wu, Yajun Ha, Akash Kumar, Shaobo Luo, Ang Li, and Shihab Mohamed. "A heterogeneous platform with GPU and FPGA for power efficient high performance computing". In: *2014 International Symposium on Integrated Circuits (ISIC)*. 2014, pp. 220–223. DOI: 10.1109/ISICIR.2014.7029447.

[31] Ashish Venkat, Harsha Basavaraj, and Dean M. Tullsen. "Composite-ISA Cores: Enabling Multi-ISA Heterogeneity Using a Single ISA". In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2019, pp. 42–55. DOI: 10.1109/HPCA.2019.00026.

[32] Evgenij Belikov, Pantazis Deligiannis, Prabhat Totoo, Malak Aljabri, and Hans-Wolfgang Loidl. "A survey of high-level parallel programming models". In: *Heriot-Watt University, Edinburgh, UK* 1.2 (2013), pp. 2–48.

[33] Leonardo Dagum and Ramesh Menon. "OpenMP: An Industry-Standard API for Shared-Memory Programming". In: *IEEE Comput. Sci. Eng.* 5.1 (Jan. 1998), pp. 46–55. DOI: 10.1109/99.660313.

[34] Peter Pacheco. *Parallel programming with MPI*. Morgan Kaufmann, 1997.

[35] Benedikt Janßen, Fynn Schwiegelshohn, Martijn Koedam, François Duhem, Leonard Masing, Stephan Werner, Christophe Huriaux, Antoine Courtay, Emilie Wheatley, Kees Goossens, Fabrice Lemonnier, Philippe Millet, Jürgen Becker, Olivier Sentieys, and Michael Hübner. "Designing applications for heterogeneous many-core architectures with the FlexTiles Platform". In: *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2015, pp. 254–261. DOI: 10.1109/SAMOS.2015.7363683.

[36] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F Brown III, and Anant Agarwal. "On-chip interconnection architecture of the tile processor". In: *IEEE micro* 27.5 (2007), pp. 15–31.

[37] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. DOI: 10.1145/1498765.1498785.

[38] Jan Gray. "GRVI Phalanx: A Massively Parallel RISC-V FPGA Accelerator Accelerator". In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2016, pp. 17–20. DOI: 10.1109/FCCM.2016.12.

[39] Bevan Baas, Zhiyi Yu, Michael Meeuwsen, Omar Sattari, Ryan Apperson, Eric Work, Jeremy Webb, Michael Lai, Tinoosh Mohsenin, Dean Truong, and Jason Cheung. "AsAP: A Fine-Grained Many-Core Platform for DSP Applications". In: *IEEE Micro* 27.2 (2007), pp. 34–45. DOI: 10.1109/MM.2007.29.

[40] Riadh Ben Abdelhamid, Yoshiki Yamaguchi, and Taisuke Boku. "A Highly-Efficient and Tightly-Connected Many-Core Overlay Architecture". In: *IEEE Access* 9 (2021), pp. 65277–65292. DOI: 10.1109/ACCESS.2021.3074171.

[41] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. "OpenPiton: An Open Source Manycore Research Framework". In: *SIGPLAN Not.* 51.4 (Mar. 2016), pp. 217–232. DOI: 10.1145/2954679.2872414.

[42] Jörg Henkel, Andreas Herkersdorf, Lars Bauer, Thomas Wild, Michael Hübner, Ravi Kumar Pujari, Artjom Grudnitsky, Jan Heisswolf, Aurang Zaib, Benjamin Vogel, Vahid Lari, and Sebastian Kobbe. "Invasive manycore architectures". In: *17th Asia and South Pacific Design Automation Conference*. 2012, pp. 193–200. DOI: 10.1109/ASPDAC.2012.6164944.

[43] Jonathan Balkind, Katie Lim, Fei Gao, Jinzheng Tu, David Wentzlaff, Michael Schaffner, Florian Zaruba, and Luca Benini. "OpenPiton+ Ariane: The first open-source, SMP Linux-booting RISC-V system scaling from one to many cores". In: *Workshop on Computer Architecture Research with RISC-V (CARRV)*. 2019, pp. 1–6.

[44] Oracle. *OpenSPARC T1*. URL: http://www.oracle.com/technetwork/systems/opensparc/opensparc-t1-page-1444609.html.

[45] Florian Zaruba and Luca Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (2019), pp. 2629–2640. DOI: 10.1109/TVLSI.2019.2926114.

[46] Young Jin Yoon, Nicola Concer, Michele Petracca, and Luca Carloni. "Virtual channels vs. multiple physical networks: A comparative analysis". In: *Design Automation Conference*. 2010, pp. 162–165.

[47] Marcelo Ruaro, Luciano L Caimi, Vinicius Fochi, and Fernando G Moraes. "Memphis: a framework for heterogeneous many-core SoCs generation and validation". In: *Design Automation for Embedded Systems* 23.3 (2019), pp. 103–122.

[48] S. Rhoads. *Plasma CPU Core*. URL: https://opencores.org/projects/plasma.

[49] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. "HERMES: An Infrastructure for Low Area Overhead Packet-Switching Networks on Chip". In: *Integr. VLSI J.* 38.1 (Oct. 2004), pp. 69–93. DOI: 10.1016/j.vlsi.2004.03.003.

[50] Daniel Petrisko, Farzam Gilani, Mark Wyse, Dai Cheol Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, Tavio Guarino, Ajay Joshi, Mark Oskin, and Michael Bedford Taylor. "BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs". In: *IEEE Micro* 40.4 (2020), pp. 93–102. DOI: 10.1109/MM.2020.2996145.

[51] Luca Benini, Eric Flamand, Didier Fuin, and Diego Melpignano. "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator". In: *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2012, pp. 983–987. DOI: 10.1109/DATE.2012.6176639.

[52] Yvain Thonnart, Pascal Vivet, and Fabien Clermidy. "A fully-asynchronous low-power framework for GALS NoC integration". In: *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. 2010, pp. 33–38. DOI: 10.1109/DATE.2010.5457239.

[53] Abbas Rahimi, Igor Loi, Mohammad Reza Kakoee, and Luca Benini. "A fully synthesizable single-cycle interconnection network for Shared-L1 processor clusters". In: *2011 Design, Automation & Test in Europe*. 2011, pp. 1–6. DOI: 10.1109/DATE.2011.5763085.

[54] F. Arnaud, S. Colquhoun, A.L. Mareau, S. Kohler, S. Jeannot, F. Hasbani, R. Paulin, S. Cremer, C. Charbuillet, G. Druais, and P. Scheer. "Technology-circuit convergence for full-SOC platform in 28 nm and beyond". In: *2011 International Electron Devices Meeting*. 2011, pp. 15.7.1–15.7.4. DOI: 10.1109/IEDM.2011.6131562.

[55] Jens Korinth, Jaco Hofmann, Carsten Heinz, and Andreas Koch. "The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems". In: *Applied Reconfigurable Computing*. Cham: Springer International Publishing, 2019, pp. 214–229.

[56] Matheus Cavalcante, Samuel Riedel, Antonio Pullini, and Luca Benini. "MemPool: A Shared-L1 Memory Many-Core Cluster with a Low-Latency Interconnect". In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2021, pp. 701–706. DOI: 10.23919/DATE51398.2021.9474087.

[57] Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. "Snitch: A Tiny Pseudo Dual-Issue Processor for Area and Energy Efficient Execution of Floating-Point Intensive Workloads". In: *IEEE Transactions on Computers* 70.11 (2021), pp. 1845–1860. DOI: 10.1109/TC.2020.3027900.

[58] Davide Giri, Kuan-Lin Chiu, Guy Eichler, Paolo Mantovani, N Chandramoorth, and Luca P Carloni. "Ariane+ NVDLA: seamless third-party IP integration with ESP". In: *Workshop on Computer Architecture Research with RISC-V (CARRV)*. 2020.

[59] Young Jin Yoon, Nicola Concer, Michele Petracca, and Luca P. Carloni. "Virtual Channels and Multiple Physical Networks: Two Alternatives to Improve NoC Performance". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.12 (2013), pp. 1906–1919. DOI: 10.1109/TCAD.2013.2276399.

[60] Cobham Gaisler. *LEON3*. URL: www.gaisler.com/index.php/products/processors/leon3.

[61] Joseph Zuckerman, Paolo Mantovani, Davide Giri, and Luca P Carloni. "Enabling Heterogeneous, Multicore SoC Research with RISC-V and ESP". In: *arXiv preprint arXiv:2206.01901* (2022).

[62] Florian Zaruba, Fabian Schuiki, and Luca Benini. "Manticore: A 4096-Core RISC-V Chiplet Architecture for Ultraefficient Floating-Point Computing". In: *IEEE Micro* 41.2 (2021), pp. 36–42. DOI: 10.1109/MM.2020.3045564.

[63] Andreas Kurth, Wolfgang Rönninger, Thomas Benz, Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, and Luca Benini. "An Open-Source Platform for High-Performance Non-Coherent On-Chip Communication". In: *IEEE Transactions on Computers* 71.8 (2022), pp. 1794–1809. DOI: 10.1109/TC.2021.3107726.

[64] Samuel Greengard. "Will RISC-V revolutionize computing?" In: *Communications of the ACM* 63.5 (2020), pp. 30–32.

[65] *RISC-V Intl.* URL: https://riscv.org/.

[66] Benjamin W. Mezger, Douglas A. Santos, Luigi Dilillo, Cesar A. Zeferino, and Douglas R. Melo. "A Survey of the RISC-V Architecture Software Support". In: *IEEE Access* 10 (2022), pp. 51394–51411. DOI: 10.1109/ACCESS.2022.3174125.

[67] Tao Lu. "A survey on risc-v security: Hardware and architecture". In: *arXiv preprint arXiv:2107.04175* (2021).

[68] Carsten Heinz, Yannick Lavan, Jaco Hofmann, and Andreas Koch. "A Catalog and In-Hardware Evaluation of Open-Source Drop-In Compatible RISC-V Softcore Processors". In: *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2019, pp. 1–8. DOI: 10.1109/ReConFig48160.2019.8994796.

[69] RISC-V. *The RISC-V Instruction Set Manual*. URL: https://riscv.org/technical/specifications/.

[70] Vectorblox. *ORCA*. URL: https://github.com/riscveval/orca-1.

[71] *PicoRV32*. URL: https://github.com/YosysHQ/picorv32.

[72] Eric Matthews and Lesley Shannon. "TAIGA: A new RISC-V soft-processor framework enabling high performance CPU architectural features". In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 2017, pp. 1–4. DOI: 10.23919/FPL.2017.8056766.

[73] *VexRiscv*. URL: https://github.com/SpinalHDL/VexRiscv.

[74] *Ibex*. URL: https://github.com/lowRISC/ibex/.

[75] Shakti. *Shakti E Class Processor*. URL: https://shakti.org.in/processors.html.

[76] *OpenHW Group CORE-V CV32E40P RISC-V IP*. URL: https://github.com/openhwgroup/cv32e40p.

[77] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications". In: *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 2017, pp. 1–8. DOI: 10.1109/PATMOS.2017.8106976.

[78] *CVA6*. URL: https://github.com/openhwgroup/cva6.

[79] Gaisler. *NOEL-V*. URL: https://www.gaisler.com/index.php/products/processors/noel-v.

[80] *Rocket Chip Generator*. URL: https://github.com/chipsalliance/rocket-chip/.

[81] Abdallah Cheikh, Stefano Sordillo, Antonio Mastrandrea, Francesco Menichelli, Giuseppe Scotti, and Mauro Olivieri. "Klessydra-T: Designing Vector Coprocessors for Multithreaded Edge-Computing Cores". In: *IEEE Micro* 41.2 (2021), pp. 64–71. DOI: 10.1109/MM.2021.3050962.

[82] *The Berkeley Out-of-Order RISC-V Processor*. URL: https://github.com/riscv-boom/riscv-boom/.

[83] codasip. *codasip H50XF core*. URL: https://codasip.com/products/codasip-risc-v-processors/.

[84] SiFive. *U7 Series*. URL: https://www.sifive.com/cores/u74.

[85] Andes. *AndesCore NX25F*. URL: http://www.andestech.com/en/products-solutions/andescore-processors/riscv-nx25f/.

[86] T-HEAD. *C910*. URL: https://www.t-head.cn/product/c910.

[87] Alexander Dörflinger, Mark Albers, Benedikt Kleinbeck, Yejun Guan, Harald Michalik, Raphael Klink, Christopher Blochwitz, Anouar Nechi, and Mladen Berekovic. "A Comparative Survey of Open-Source Application-Class RISC-V Processor Implementations". In: *Proceedings of the 18th ACM International Conference on Computing Frontiers*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 12–20. DOI: 10.1145/3457388.3458657.

[88] Sparsh Mittal et al. "A survey of accelerator architectures for 3D convolution neural networks". In: *Journal of Systems Architecture* 115 (2021), p. 102041.

[89] Lester Kalms, Pedram Amini Rad, Muhammad Ali, Arsany Iskander, and Diana Göhringer. "A parametrizable high-level synthesis library for accelerating neural networks on fpgas". In: *Journal of Signal Processing Systems* 93.5 (2021), pp. 513–529.

[90] Deepak Ghimire, Dayoung Kil, and Seong-heum Kim. "A Survey on Efficient Convolutional Neural Networks and Hardware Acceleration". In: *Electronics* 11.6 (2022), p. 945.

[91] Ahmed Kamaleldin, Ensieh Aliagha, Aman Batra, Michael Wiemeler, Thomas Kaiser, and Diana Göhringer. "Hardware/Software Co-design of 2D THz SAR Imaging for FPGA-based Systems-on-Chip". In: *2022 Fifth International Workshop on Mobile Terahertz Systems (IWMTS)*. 2022, pp. 1–5. DOI: 10.1109/IWMTS54901.2022.9832447.

[92] Ahmed Kamaleldin, Jonas Wagner, Ilona Rolfes, Jan Barowski, and Diana Goehringer. "Hardware/Software Co-design for the Signal Processing of Dielectric Materials Characterization". In: *2020 Third International Workshop on Mobile Terahertz Systems (IWMTS)*. 2020, pp. 1–6. DOI: 10.1109/IWMTS49292.2020.9166402.

[93] Aman Batra, Ahmed Kamaleldin, Lee Ye Zhen, Michael Wiemeler, Diana Göhringer, and Thomas Kaiser. "FPGA-Based Acceleration of THz SAR Imaging". In: *2021 Fourth International Workshop on Mobile Terahertz Systems (IWMTS)*. 2021, pp. 1–5. DOI: 10.1109/IWMTS51331.2021.9486819.

[94] Lester Kalms, Maximilian Hajduk, and Diana Göhringer. "Efficient Pattern Recognition Algorithm Including a Fast Retina Keypoint FPGA Implementation". In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019, pp. 121–128. DOI: 10.1109/FPL.2019.00028.

[95] Lester Kalms, Ariel Podlubne, and Diana Göhringer. "HiFlipVX: An Open Source High-Level Synthesis FPGA Library for Image Processing". In: *Applied Reconfigurable Computing*. Ed. by Christian Hochberger, Brent Nelson, Andreas Koch, Roger Woods, and Pedro Diniz. Cham: Springer International Publishing, 2019, pp. 149–164.

[96] Lester Kalms, Hassan Ibrahim, and Diana Göhringer. "Full-HD Accelerated and Embedded Feature Detection Video System with 63fps using ORB for FREAK". In: *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2018, pp. 1–6. DOI: 10.1109/RECONFIG.2018.8641706.

[97] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. "The architectural implications of autonomous driving: Constraints and acceleration". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 2018, pp. 751–766.

[98] William J. Dally, Yatish Turakhia, and Song Han. "Domain-Specific Hardware Accelerators". In: *Commun. ACM* 63.7 (June 2020), pp. 48–57. DOI: 10.1145/3361682.

[99] Hamed Tabkhi, Robert Bushey, and Gunar Schirner. "Function-Level Processor (FLP): A High Performance, Minimal Bandwidth, Low Power Architecture for Market-Oriented MPSoCs". In: *IEEE Embedded Systems Letters* 6.4 (2014), pp. 65–68. DOI: 10.1109/LES.2014.2327114.

[100] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. "Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication". In: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 65–77. DOI: 10.1145/3490422.3502357.

[101] Andreas Bytyn, Rainer Leupers, and Gerd Ascheid. "ConvAix: An Application-Specific Instruction-Set Processor for the Efficient Acceleration of CNNs". In: *IEEE Open Journal of Circuits and Systems* 2 (2021), pp. 3–15. DOI: 10.1109/OJCAS.2020.3037758.

[102] Wei Mao, Kai Li, Quan Cheng, Liuyao Dai, Boyu Li, Xinang Xie, He Li, Longyang Lin, and Hao Yu. "A Configurable Floating-Point Multiple-Precision Processing Element for HPC and AI Converged Computing". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 30.2 (2022), pp. 213–226. DOI: 10.1109/TVLSI.2021.3128435.

[103] Emilio G. Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. "An analysis of accelerator coupling in heterogeneous architectures". In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6. DOI: 10.1145/2744769.2744794.

[104] G. Ezer. "Xtensa with user defined DSP coprocessor microarchitectures". In: *Proceedings 2000 International Conference on Computer Design*. 2000, pp. 335–342. DOI: 10.1109/ICCD.2000.878305.

[105] Ho-Cheung Ng, Cheng Liu, and Hayden Kwok-Hay So. "A soft processor overlay with tightly-coupled FPGA accelerator". In: *arXiv preprint arXiv:1606.06483* (2016).

[106] Wenqi Lou, Chao Wang, Lei Gong, and Xuehai Zhou. "RV-CNN: flexible and efficient instruction set for CNNs based on RISC-V processors". In: *International Symposium on Advanced Parallel Processing Technologies*. Springer. 2019, pp. 3–14.

[107] Muhammad Ali, Matthias von Ameln, and Diana Goehringer. "Vector Processing Unit: A RISC-V based SIMD Co-processor for Embedded Processing". In: *2021 24th Euromicro Conference on Digital System Design (DSD)*. 2021, pp. 30–34. DOI: 10.1109/DSD53832.2021.00014.

[108] Joseph Zuckerman, Davide Giri, Jihye Kwon, Paolo Mantovani, and Luca P. Carloni. "Cohmeleon: Learning-Based Orchestration of Accelerator Coherence in Heterogeneous SoCs". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 350–365. DOI: 10.1145/3466752.3480065.

[109] Ali Farahani, Hakem Beithollahi, Mahmood Fathi, and Reza Barangi. "CNNX: A Low Cost, CNN Accelerator for Embedded System in Vision at Edge". In: *Arabian Journal for Science and Engineering* (2022), pp. 1–9.

[110] Yvan Tortorella, Luca Bertaccini, Davide Rossi, Luca Benini, and Francesco Conti. "RedMulE: A Compact FP16 Matrix-Multiplication Accelerator for Adaptive Deep Learning on RISC-V-Based Ultra-Low-Power SoCs". In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2022, pp. 1099–1102. DOI: 10.23919/DATE54114.2022.9774759.

[111] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. "Charm: A composable heterogeneous accelerator-rich microprocessor". In: *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*. 2012, pp. 379–384.

[112] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. "Accelerator-rich architectures: Opportunities and progresses". In: *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE. 2014, pp. 1–6.

[113] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Chunyue Liu, Glenn Reinman, and Yi Zou. "AXR-CMP: Architecture support in accelerator-rich CMPs". In: *2nd Workshop on SoC Architecture, Accelerators and Workloads*. 2011.

[114] Gianluca Bellocchi, Alessandro Capotondi, Francesco Conti, and Andrea Marongiu. "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.

[115] Guy Eichler, Luca Piccolboni, Davide Giri, and Luca P. Carloni. "MasterMind: Many-Accelerator SoC Architecture for Real-Time Brain-Computer Interfaces". In: *2021 IEEE 39th International Conference on Computer Design (ICCD)*. 2021, pp. 101–108. DOI: 10.1109/ICCD53106.2021.00027.

[116] Gerhard Fettweis, Mattis Hassler, Robert Wittig, Emil Matus, Stefan Damjancevic, Sebastian Haas, Friedrich Pauls, Seungseok Nam, and Nairuhi Grigoryan. "A Low-Power Scalable Signal Processing Chip Platform for 5G and Beyond - Kachel". In: *2019 53rd Asilomar Conference on Signals, Systems, and Computers*. 2019, pp. 896–900. DOI: 10.1109/IEEECONF44664.2019.9048785.

[117] Emilio G. Cota, Paolo Mantovani, Michele Petracca, Mario R. Casu, and Luca P. Carloni. "Accelerator Memory Reuse in the Dark Silicon Era". In: *IEEE Computer Architecture Letters* 13.1 (2014), pp. 9–12. DOI: 10.1109/L-CA.2012.29.

[118] Michael Lyons, Mark Hempstead, Gu-Yeon Wei, and David Brooks. "The Accelerator Store framework for high-performance, low-power accelerator-based systems". In: *IEEE Computer Architecture Letters* 9.2 (2010), pp. 53–56. DOI: 10.1109/L-CA.2010.16.

[119] Paolo Mantovani, Emilio G. Cota, Christian Pilato, Giuseppe Di Guglielmo, and Luca P. Carloni. "Handling Large Data Sets for High-Performance Embedded Applications in Heterogeneous Systems-on-Chip". In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '16. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2016. DOI: 10.1145/2968455.2968509.

[120] Masoud Dehyadegari, Andrea Marongiu, Mohammad Reza Kakoee, Siamak Mohammadi, Naser Yazdani, and Luca Benini. "Architecture Support for Tightly-Coupled Multi-Core Clusters with Shared-Memory HW Accelerators". In: *IEEE Transactions on Computers* 64.8 (2015), pp. 2132–2144. DOI: 10.1109/TC.2014.2360522.

[121] Gianluca Bellocchi, Alessandro Capotondi, Francesco Conti, and Andrea Marongiu. "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.

[122] Bin Li, Zhen Fang, and Ravi Iyer. "Template-based memory access engine for accelerators in SoCs". In: *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011)*. 2011, pp. 147–153. DOI: 10.1109/ASPDAC.2011.5722175.

[123] Davide Giri, Paolo Mantovani, and Luca P. Carloni. "Accelerators and Coherence: An SoC Perspective". In: *IEEE Micro* 38.6 (2018), pp. 36–45. DOI: 10.1109/MM.2018.2877288.

[124] Davide Giri, Paolo Mantovani, and Luca P. Carloni. "NoC-Based Support of Heterogeneous Cache-Coherence Models for Accelerators". In: *2018 Twelfth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*. 2018, pp. 1–8. DOI: 10.1109/NOCS.2018.8512153.

[125] Davide Giri, Paolo Mantovani, and Luca P. Carloni. "Runtime Reconfigurable Memory Hierarchy in Embedded Scalable Platforms". In: *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ASPDAC '19. Tokyo, Japan: Association for Computing Machinery, 2019, pp. 719–726. DOI: 10.1145/3287624.3288755.

[126] Christian Pilato, Qirui Xu, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. "On the Design of Scalable and Reusable Accelerators for Big Data Applications". In: *Proceedings of the ACM International Conference on Computing Frontiers*. CF '16. Como, Italy: Association for Computing Machinery, 2016, pp. 406–411. DOI: 10.1145/2903150.2906141.

[127] Christian Pilato, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. "System-Level Optimization of Accelerator Local Memory for Heterogeneous Systems-on-Chip". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36.3 (2017), pp. 435–448. DOI: 10.1109/TCAD.2016.2611506.

[128] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Chunyue Liu, and Glenn Reinman. "BiN: A Buffer-in-NUCA Scheme for Accelerator-Rich CMPs". In: *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*. ISLPED '12. Redondo Beach, California, USA: Association for Computing Machinery, 2012, pp. 225–230. DOI: 10.1145/2333660.2333715.

[129] Emilio G. Cota, Paolo Mantovani, and Luca P. Carloni. "Exploiting Private Local Memories to Reduce the Opportunity Cost of Accelerator Integration". In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS '16. Istanbul, Turkey: Association for Computing Machinery, 2016. DOI: 10.1145/2925426.2926258.

[130] Harshal G Hayatnagarkar, MV Arunkumar, and Bhimsen Padalkar. "Reconfigurable Domain-specific Architectures in the post-Moore's Law World: Implications for Software Engineering". In: *Preprint* (2020).

[131] Xilinx. *UltraScale Architecture Configuration*. URL: https://docs.xilinx.com/v/u/en-US/ug570-ultrascale-configuration.

[132] Xilinx. *Partial Reconfiguration UG909 v2019.1*. URL: https://docs.xilinx.com/v/u/2019.1-English/ug909-vivado-partial-reconfiguration.

[133] Kizheppatt Vipin and Suhaib A Fahmy. "FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications". In: *ACM Computing Surveys (CSUR)* 51.4 (2018), pp. 1–39.

[134] Ahmed Kamaleldin, Sherif Hosny, Khaled Mohamed, Mostafa Gamal, Abdelrhman Hussien, Eslam Elnader, Ahmed Shalash, Abdelfattah M. Obeid, Yehea Ismail, and Hassan Mostafa. "A reconfigurable hardware platform implementation for software defined radio using dynamic partial reconfiguration on Xilinx Zynq FPGA". In: *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2017, pp. 1540–1543. DOI: 10.1109/MWSCAS.2017.8053229.

[135] Sherif Hosny, Eslam Elnader, Mostafa Gamal, Abdelrhman Hussien, Ahmed H. Khalil, and Hassan Mostafa. "A Software Defined Radio Transceiver Based on Dynamic Partial Reconfiguration". In: *2018 New Generation of CAS (NGCAS)*. 2018, pp. 158–161. DOI: 10.1109/NGCAS.2018.8572253.

[136] Julien Mazuet, Michel Narozny, Catherine Dezan, and Jean-Philippe Diguet. "A Seamless DFT/FFT Self-Adaptive Architecture for Embedded Radar Applications". In: *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. 2020, pp. 115–120. DOI: 10.1109/FPL50879.2020.00029.

[137] Marie Nguyen, Robert Tamburo, Srinivasa Narasimhan, and James C. Hoe. "Quantifying the Benefits of Dynamic Partial Reconfiguration for Embedded Vision Applications". In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019, pp. 129–135. DOI: 10.1109/FPL.2019.00029.

[138] Khaled Khatib, Mostafa Ahmed, Ahmed Kamaleldin, Mohamed Abdelghany, and Hassan Mostafa. "Dynamically reconfigurable power efficient security for Internet of Things devices". In: *2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. 2018, pp. 1–4. DOI: 10.1109/MOCAST.2018.8376645.

[139] BC Manjith and R Dhanalakshmi. "Enabling self-adaptability of small scale and large scale security systems using dynamic partial reconfiguration". In: vol. 12. Springer, 2021, pp. 9387–9403.

[140] Fernando Georgel Bîrleanu and Nicu Bizon. "Lightweight cryptography for Internet of Things using FPGA-based Design with Partial Reconfiguration". In: *2020 12th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*. 2020, pp. 1–7. DOI: 10.1109/ECAI50035.2020.9223213.

[141] Hanaa M Hussain, Khaled Benkrid, Ali Ebrahim, Ahmet T Erdogan, and Huseyin Seker. "Novel dynamic partial reconfiguration implementation of k-means clustering on FPGAs: Comparative results with GPPs and GPUs". In: vol. 2012. Hindawi Limited London, UK, United Kingdom, 2012, pp. 1–1.

[142] Salma Hassan, Sameh Attia, Khaled Nabil Salama, and Hassan Mostafa. "EANN: Energy adaptive neural networks". In: vol. 9. 5. MDPI, 2020, p. 746.

[143] Arturo Pérez, Alfonso Rodríguez, Andrés Otero, David González Arjona, Álvaro Jiménez-Peralo, Miguel Ángel Verdugo, and Eduardo De La Torre. "Run-Time Reconfigurable MPSoC-Based On-Board Processor for Vision-Based Space Navigation". In: *IEEE Access* 8 (2020), pp. 59891–59905. DOI: 10.1109/ACCESS.2020.2983308.

[144] Alfonso Rodríguez, Andrés Otero, Marco Platzner, and Eduardo de la Torre. "Exploiting Hardware-Based Data-Parallel and Multithreading Models for Smart Edge Computing in Reconfigurable FPGAs". In: *IEEE Transactions on Computers* 71.11 (2022), pp. 2903–2914. DOI: 10.1109/TC.2021.3107196.

[145] Rafael Zamacola, Andrés Otero, and Eduardo de la Torre. "Multi-grain reconfigurable and scalable overlays for hardware accelerator composition". In: *Journal of Systems Architecture* 121 (2021), p. 102302.

[146] Suhaib A. Fahmy. "Design Abstraction for Autonomous Adaptive Hardware Systems on FPGAs". In: *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. 2018, pp. 142–147. DOI: 10.1109/AHS.2018.8541489.

[147] Luca Pezzarossa, Andreas Toftegaard Kristensen, Martin Schoeberl, and Jens Sparso. "Can real-time systems benefit from dynamic partial reconfiguration?" In: *2017 IEEE Nordic Circuits and Systems Conference, and International Symposium of System-on-Chip*. 2017, pp. 1–6. DOI: 10.1109/NORCHIP.2017.8124984.

[148] Kizheppatt Vipin and Suhaib A. Fahmy. "A high speed open source controller for FPGA Partial Reconfiguration". In: *2012 International Conference on Field Programmable Technology*. 2012, pp. 61–66. DOI: 10.1109/FPT.2012.6412113.

[149] Kizheppatt Vipin and Suhaib A. Fahmy. "ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq". In: *IEEE Embedded Systems Letters* 6.3 (2014), pp. 41–44. DOI: 10.1109/LES.2014.2314390.

[150] Stefano Di Carlo, Paolo Prinetto, Pascal Trotta, and Jan Andersson. "A portable open-source controller for safe Dynamic Partial Reconfiguration on Xilinx FPGAs". In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. 2015, pp. 1–4. DOI: 10.1109/FPL.2015.7294002.

[151] Luca Pezzarossa, Martin Schoeberl, and Jens Sparsø. "A Controller for Dynamic Partial Reconfiguration in FPGA-Based Real-Time Systems". In: *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*. 2017, pp. 92–100. DOI: 10.1109/ISORC.2017.3.

[152] Luis Andres Cardona and Carles Ferrer. "AC_ICAP: A flexible high speed ICAP controller". In: *International Journal of Reconfigurable Computing* 2015 (2015).

[153] Christian Kohn. "Partial reconfiguration of a hardware accelerator on zynq-7000 all programmable soc devices". In: *Xilinx, XAPP1159 (v1. 0)* (2013).

[154] Muhammed Al Kadi, Patrick Rudolph, Diana Goehringer, and Michael Huebner. "Dynamic and partial reconfiguration of Zynq 7000 under Linux". In: *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. 2013, pp. 1–5. DOI: 10.1109/ReConFig.2013.6732279.

[155] Benedikt Janßen, Pascal Zimprich, and Michael Hübner. "A dynamic partial reconfigurable overlay concept for PYNQ". In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 2017, pp. 1–4. DOI: 10.23919/FPL.2017.8056786.

[156] Alex R. Bucknall, Shanker Shreejith, and Suhaib A. Fahmy. "Build Automation and Runtime Abstraction for Partial Reconfiguration on Xilinx Zynq UltraScale+". In: *2020 International Conference on Field-Programmable Technology (ICFPT)*. 2020, pp. 215–220. DOI: 10.1109/ICFPT51103.2020.00037.

[157] Andreas Kurth, Pirmin Vogel, Alessandro Capotondi, Andrea Marongiu, and Luca Benini. "HERO: Heterogeneous embedded research platform for exploring RISC-V manycore accelerators on FPGA". In: *arXiv preprint arXiv:1712.06497* (2017).

[158] Farhad Merchant, Dominik Sisejkovic, Lennart M. Reimann, Kirthihan Yasotharan, Thomas Grass, and Rainer Leupers. "ANDROMEDA: An FPGA Based RISC-V MPSoC Exploration Framework". In: *2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID)*. 2021, pp. 270–275. DOI: 10.1109/VLSID51830.2021.00051.

[159] Süleyman Savas, Zain Ul-Abdin, and Tomas Nordström. "A framework to generate domain-specific manycore architectures from dataflow programs". In: *Microprocessors and microsystems* 72 (2020), p. 102908.

[160] Johannes Ax, Gregor Sievers, Julian Daberkow, Martin Flasskamp, Marten Vohrmann, Thorsten Jungeblut, Wayne Kelly, Mario Porrmann, and Ulrich Rückert. "CoreVA-MPSoC: A Many-Core Architecture with Tightly Coupled Shared and Local Data Memories". In: *IEEE Transactions on Parallel and Distributed Systems* 29.5 (2018), pp. 1030–1043. DOI: 10.1109/TPDS.2017.2785799.

[161] Wilson José, Horácio Neto, and Mário Véstias. "A Many-Core Co-Processor for Embedded Parallel Computing on FPGA". In: *2015 Euromicro Conference on Digital System Design*. 2015, pp. 539–542. DOI: 10.1109/DSD.2015.23.

[162] Mahmoud A. Elmohr, Ahmed S. Eissa, Moamen Ibrahim, Mostafa Khamis, Sameh El-Ashry, Ahmed Shalaby, Mohamed AbdElsalam, and M. Watheq El-Kharashi. "RVNoC: A Framework for Generating RISC-V NoC-Based MPSoC". In: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 2018, pp. 617–621. DOI: 10.1109/PDP2018.2018.00103.

[163] Andreas Olofsson, Tomas Nordström, and Zain Ul-Abdin. "Kickstarting high performance energy-efficient manycore architectures with Epiphany". In: *2014 48th Asilomar Conference on Signals, Systems and Computers*. 2014, pp. 1719–1726. DOI: 10.1109/ACSSC.2014.7094761.

[164] Benôıt Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benôıt Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, and Thierry Strudel. "A clustered manycore processor architecture for embedded and accelerated applications". In: *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. 2013, pp. 1–6. DOI: 10.1109/HPEC.2013.6670342.

[165] PULP-Platform. *AXI SystemVerilog Modules for High-Performance On-Chip Communication*. URL: https://github.com/pulp-platform/axi.

[166] Andreas Traber, Michael Gautschi, and Pasquale David Schiavone. *RI5CY: User Manual*. URL: https://pulp-platform.org/docs/ri5cy_user_manual.pdf.

[167] PULP-Platform. *PULP RISC-V GNU Compiler Toolchain*. URL: https://github.com/pulp-platform/pulp-riscv-gnutoolchain.

[168] Xilinx. *Xilinx Virtex UltraScale+ FPGA VCU118 Evaluation Kit*. URL: https://www.xilinx.com/products/boards-and-kits/vcu118.html.

[169] Xilinx. *Vivado Design Suit*. URL: https://www.xilinx.com/products/design-tools/vivado.html.

[170] ESP. *ESP: The Open-Source SoC Platform*. URL: https://github.com/sld-columbia/esp.

[171] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. "PULP-NN: A Computing Library for Quantized Neural Network inference at the edge on RISC-V Based Parallel Ultra Low Power Clusters". In: *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2019, pp. 33–36. DOI: 10.1109/ICECS46596.2019.8965067.

[172] PULP-Platform. *PULP-NN: Enabling QNN inference on PULP*. URL: https://github.com/pulp-platform/pulp-nn.

[173] Xilinx. *Fast Fourier Transform v9.0 (PG109)*. URL: https://www.xilinx.com/support/documentation/ipdocumentation/xfft/v90/pg109-xfft.pdf.

[174] Microchip. *PolarFire SoC FPGAs (Architecture, Applications, Scurity Features, Design Environment, Design Hardware)*. URL: /https://www.microsemi.com/document-portal/doc_view/1244582-polarfire-soc-brochure.

[175] Nguyen Dao, Andrew Attwood, Bea Healy, and Dirk Koch. "FlexBex: A RISC-V with a Reconfigurable Instruction Extension". In: *2020 International Conference on Field-Programmable Technology (ICFPT)*. 2020, pp. 190–195. DOI: 10.1109/ICFPT51103.2020.00034.

[176] Pasquale Davide Schiavone, Davide Rossi, Alfio Di Mauro, Frank K. Gürkaynak, Timothy Saxe, Mao Wang, Ket Chong Yap, and Luca Benini. "Arnold: An eFPGA-Augmented RISC-V SoC for Flexible and Low-Power IoT End Nodes". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 29.4 (2021), pp. 677–690. DOI: 10.1109/TVLSI.2021.3058162.

[177] Xilinx. *AXI HWICAP*. URL: https://www.xilinx.com/products/intellectual-property/axi_hwicap.html.

[178] Xilinx. *AXI DMA LogiCORE IP Product Guide*. URL: https://docs.xilinx.com/r/en-US/pg021_axi_dma.

[179] TUD-ADS. *HiFlipVX: Open Source High-Level Synthesis FPGA Library for Image Processing*. URL: https://github.com/TUD-ADS/HiFlipVX.

Student Work

[STD1] Marcus Gottschalk. “Investigating Multicore RISC-V Architectures”. Studienarbeit. May 2019, pp. 1–86.

[STD2] Florian Schuster. “Development of a 64-bit RISC-V based Multi-core Shared Memory System”. Studienarbeit. Mar. 2021, pp. 1–53.

[STD3] Martin Thümmler. “Realization of a Self-Adaptive RISC-V System for Image Processing Algorithms”. Project Work. Apr. 2021, pp. 1–46.

[STD4] Oguzhan Türk. “Accelerating Signal Processing Kernels on RISC-V based MPSoC”. Project Work. Mar. 2022, pp. 1–50.

[STD5] Ambuja Vinayak Rashinkar. “Accelerating Machine Learning Kernels on RISC-V based MPSoC”. Project Work. Mar. 2022, pp. 1–48.