Technical University of Crete

Doctoral Thesis

UniLogic (Unified Logic): A Scalable Architecture for Increased Programmability in Highly Parallel Reconfigurable Systems

Author:
Aggelos D. Ioannou

Supervisor:
Prof. Apostolos Dollas

A thesis submitted in fulfillment of the requirements for the Doctor of Philosophy degree of Electrical and Computer Engineering

in the School of Electrical and Computer Engineering

Microprocessor and Hardware Lab

February 28, 2020

Doctoral Thesis Committee

Thesis Statement

“The state-of-the-art Supercomputers utilize many-core accelerators instead of energy-efficient FPGAs, mainly due to low programmability of multi-FPGA environments. This thesis provides and evaluates the scalable UNILOGIC architecture, which can significantly improve the programmability of multi-FPGA environments without sacrificing performance, and thus proves that FPGA technology can constitute a viable alternative to tackle today’s HPC energy challenges. ”

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

DOCTORAL THESIS

UniLogic (Unified Logic):
A Scalable Architecture for Increased Programmability in Highly Parallel Reconfigurable Systems

by Aggelos D. Ioannou

One of the main characteristics of HPC applications is that they become increasingly performance and power demanding, pushing HPC systems to their limits. Existing HPC systems have not yet reached exascale performance mainly due to power limitations. Extrapolating from today's top HPC systems, about 100-200 MWatts would be required in order to sustain an exaflop-level of performance. A promising solution for tackling power limitations, is the deployment of energy-efficient reconfigurable resources (in the form of FPGAs) tightly integrated with conventional CPUs. However, current FPGA tools and programming environments are optimized for accelerating a single application or even task on a single FPGA device. In this thesis we present UNILOGIC (Unified Logic), a novel HPC-tailored parallel architecture that efficiently incorporates FPGAs. UNILOGIC adopts the Partitioned Global Address Space (PGAS) model, and extends it to include hardware accelerators, i.e. tasks implemented on the reconfigurable resources. The main advantages of UNILOGIC are that (i) the hardware accelerators can be accessed directly by any processor in the system, and (ii) the hardware accelerators can access any memory location in the system. In this way, the proposed architecture offers a unified environment where all the reconfigurable resources can be seamlessly used by any processor/operating system. The UNILOGIC architecture also provides hardware virtualization of the reconfigurable logic so that the hardware accelerators can be shared among multiple applications or tasks. The FPGA layer of the architecture is implemented by splitting its reconfigurable resources into *(i)* a static partition, which provides the PGAS-related communication infrastructure, and *(ii)* fixed-size and dynamically reconfigurable slots that can be programmed and accessed independently or combined together so as to support both fine and coarse grain reconfiguration. Finally, the UNILOGIC architecture has been evaluated on a custom prototype that consists of two 1U chassis, each of which hosts eight interconnected daughter boards, called Quad-FPGA Daughter Boards (QFDBs); each QFDB supports four tightly coupled Xilinx Zynq Ultrascale+ MPSoCs as well as 64 Gigabytes of DDR4 memory, and thus, the prototype features a total of 64 Zynq MPSoCs and 1 Terabyte of memory. We tuned and evaluated the UNILOGIC prototype using both low-level (baremetal) performance tests, as well as two popular real-world HPC applications, one compute-intensive and one data-intensive. Our evaluation shows that UNILOGIC offers impressive performance that ranges from being 3 to 400 times faster and 46 to 370 times more energy efficient compared to conventional parallel systems utilizing only high-end CPUs, while it also outperforms GPUs by a factor ranging from 6 to 20 times in terms of time to solution, and from 8 to 20 times in terms of energy to solution.

Acknowledgements

I would like to thank my supervisor Prof. Apostolos Dollas for his guidance during my work, and for helping me focus on research using the correct point of view, the proper perspective. Also prof. Ioannis Papaefstathiou who productively supervised my work from the very first stages on to its fulfillment, and prosperously guided me throughout my efforts. Likewise prof. Dionisios Pnevmatikatos for his helpful advises at various points of my work. Furthermore, all the members of my doctoral thesis committee, for their participation and beneficial advising.

I would also like to express my sincere gratitude to Prof. Manolis G. H. Katevenis for the inspiration and encouragement. I would also like to thank Dr. Iakovos Mavroidis for exchanging ideas as well as discussing towards solutions during my research, while effectively coordinating the associated ECOSCALE project, and for trusting me to fulfill the corresponding research and development. I am also thankful to Dr. Manolis Marazakis for our helpful discussions throughout this period.

Furthermore Konstantinos Harteros for greatly helping me to quickly come on track in the first stages of my work, and Dr. Fabien Chaix for his ideas as well as crucial help and cooperation through various phases of the research, design and development process. For that purpose also Nikolaos Dimou, Pantelis Xirouchakis, Vassilis Flouris and Dr. Vassilis Papaefstathiou for their collaboration on design and verification methodologies.

Dr. Kostas Georgopoulos and PhD candidate Pavlos Malakonakis offered their invaluable help and cooperation, mainly for the operating system support and associated drivers. Also for greatly contributing during the investigation concerning the co-design process for optimizing acceleration under the UNILOGIC architecture.

Also the Telecommunication Systems Institute (TSI), Chania, Greece where most of the work was hosted and funded. Likewise, the CARV Laboratory of the Institute of Computer Science (CARV-ICS) at FORTH for the collaboration on various aspects of the related research and development.

Last but not least I would like to express my deepest gratitude to my father, who would be truly delighted, my mother who has always been helpful and supportive, and all my family. I feel I should name my brother George for sharing his excitement throughout this period, while both my brothers George and Gerasimos showed their confidence in me. Also my friends and colleagues for supporting me all the way, despite the very limited time I could share during this period. Not to omit, my co-athletes for offering me joyful and relaxing moments at the rare occasion I could train with them.

This work was supported by the European Commission under the H2020 Programme and the ECOSCALE project (grant agreement 671632).

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

FPGA	**F**ield **P**rogrammable **G**ate **A**rray
CPU	**C**entral **P**rocessor **U**nit
GPU	**G**raphic **P**rocessor **U**nit
TPU	**T**ensor **P**rocessor **U**nit
SoC	**S**ystem on **C**hip
MPSoC	**M**ulti **P**rocessor **S**ystem on **C**hip
NoC	**N**etwork on **C**hip
PS	**P**rocessing **S**ystem
PL	**P**rogramming **L**ogic
DSP	**D**igital **S**ignal **P**rocessor
FF	**F**lip **F**lops
LUT	**L**ook **U**p **T**able
CLB	**C**onfigurable **L**ogic **B**lock
AXI	**A**dvanced e**X**tensible **I**nterface
ICAP	**I**nternal **C**onfiguration **A**ccess **P**ort
PCAP	**P**rocesser
Configuration	**A**ccess
Port **FSBL**	**F**irst
Stage	**B**oot
Loader **QFDB**	**Q**uad **F**PGA **D**aughter **B**oard
PCB	**P**rinted **C**ircuit **B**oard
BMC	**B**aseboard **M**anagement **C**ontroller
HLS	**H**igh **L**evel **S**ynthesis
SDK	**S**oftware **D**evelopment **K**it
RAM	**R**andom **A**ccess **M**emory
D-RAM	**D**ynamic **R**andom **A**ccess **M**emory
B-RAM	**B**lock **R**andom **A**ccess **M**emory
CAM	**C**ontent

Addressable	**M**emory
DDR	**D**ouble **D**ata **R**ate
SODIMM	**S**mall **O**utline **D**ual **I**n-line **M**emory **M**odule
QSPI	**Q**uad **S**erial **P**eripheral **I**nterface
GAS	**G**lobal **A**ddress **S**pace
PGAS	**P**artitioned **G**lobal **A**ddress **S**pace
OS	**O**perating **S**ystem
CDD	**C**haracter **D**evice **D**river
MPI	**M**essage **P**assing **I**nterface
HPC	**H**igh **P**erformance **C**omputing
DMA	**D**irect **M**emory **A**ccess
RDMA	**R**emote **D**irect **M**emory **A**ccess
FLOPS	**F**loating **P**oint **OP**erations per **S**econd
GbE	**G**iga-bit **E**thernet
HSSL	**H**igh **S**peed **S**erial **Link**
C2C	**C**hip to **C**hip
RTT	**R**ound **T**rip **T**ime
RGMII	**R**educed **G**igabit **M**edia-**I**dependent **I**nterface
SFP	**S**mall **Form**-factor **P**luggable
RFU	**R**eserved for **F**uture **U**se

Dedicated to my parents, family and friends…

Chapter 1

Introduction

High Performance Computing (HPC) has traditionally been a driver of advanced technology in computing hardware. The wide spread of computational methods in many scientific fields has propelled HPC systems even further, and current deployment of HPC machines can support somewhere in the order of 100 petaflops, i.e.10^{17} flops, with an immediate goal to deploy exaflop-scale machines in the near future.

One of the main challenges in building larger HPC machines is power efficiency. Scaling processor clock speed is infeasible due to power envelope restrictions, and adding more and more processors, while feasible, soon hits the power consumption wall. To increase efficiency and achieve high performance at a lower power expenditure, the scientific community, performing a paradigm shift, introduced the concept of *heterogeneity* into the domain of HPC. As a result, currently, most HPC systems are comprised of conventional CPUs, tightly integrated with diverse energy-efficient processing elements, such as the traditional Vector processors and more recently GPUs; such processing elements have the advantage of being optimized for throughput and performance per Watt when they perform bulk computations.

1.1 Motivation

An even more promising approach is to include reconfigurable units in the system. Research shows that for a wide range of applications, reconfigurable (typically streaming dataflow) accelerators can be crafted and, even if sometimes they do not exceed the performance of classical approaches by much, they are significantly more energy-efficient. As a result, FPGAs are now becoming competitors to many-core accelerators, such as GPUs and Vector processors. However, a major obstacle in adopting FPGAs in HPC is their limited programmability and the maturity of the development tools, that discourage application developers from utilizing them. Historically, FPGAs are used as (reprogrammable) hardware blocks, and more recently as standalone systems (SoCs), so the tools and programming environments are usually optimized for compiling and running efficiently a single application on a single FPGA. Moreover, there is increasing –but still limited– support for a number of FPGA features necessary for an efficient HPC system, such as, managing the FPGA through an Operating System (OS), executing an application in-parallel on multiple FPGAs, and running multiple applications on the same FPGA. Hence, what is missing is the programming flexibility that is taken for granted in multi-core systems but is not well supported on FPGA-based environments.

This relates closely to a series of open problems, regarding the deployment of FPGAs in the HPC domain. Currently FPGAs, although extensively used in cloud infrastructure and data centers, are still not deployed in the HPC domain, mainly as they are difficult to use. This difficulty comes from the aforementioned **(i)** from their low programmability, with current tools mainly targeted on optimizing a single application on a single FPGA device. Furthermore, **(ii)** different FPGA vendors, as well as different FPGA families of the same vendor and varying FPGA sizes per family, affect the HW realization of task accelerators. Very specialized HW as well as SW skills are thus required, which correspond to great amounts of effort. What is more, **(iii)** the offered solutions are not generic, and correspond to very specific workloads, while they cannot be reprogrammed and reused in order to provide broader solutions. Another important issue is that **(iv)** still FPGAs are usually tightly coupled to traditional CPUs and do not leverage their competence for efficient inter-FPGA connectivity. FPGAs need to become disaggregated from the CPU/GPU infrastructure in order to prove fully effective in HPC infrastructures. Finally, **(v)** there are no reported solutions to the author's knowledge that present evaluations of such proposed, generic systems in realized hardware platforms. Due to all the aforementioned issues, there exists a great concern on the suitability of the FPGA alternative for the already heterogeneous HPC domain.

To tackle those issues we introduce the UNILOGIC architecture and framework that incorporates FPGAs within an HPC platform seamlessly and efficiently.

Not only does it provide a parallel hardware execution environment for applications running on multiple FPGAs, but also offers a simple programming environment and system software for using efficiently and seamlessly the reconfigurable resources of multiple FPGAs. Our motivation and ultimate goal is to develop a system that can seamlessly utilize all FPGA resources as if they were all packaged in a single chip that offers massive amount of resources. Doing so in a straightforward and efficient manner, promotes performance while in parallel provides ease of programmability.

1.2 Thesis Statement

This thesis demonstrates that the FPGA technology can and should be deployed in the HPC domain. That is, FPGA devices are a viable alternative to elevate the heterogeneous HPC infrastructure, if offered the proper architecture. The proposed and realized UNILOGIC architecture offers a novel approach that incorporates a low latency, distributed interconnection infrastructure, along with resource virtualization and partial reconfiguration, so that hundreds or more of FPGA devices can be used in parallel, proving to be highly efficient and scalable.

In particular, this thesis showcases that, based on the novel UNILOGIC architecture, we can provide an easily programmable, efficient and scalable multi-processor-like FPGA infrastructure, where multiple hardware accelerators are spread around the system. These can be easily orchestrated by an application and/or runtime system, and highly parallelize the execution on the available reconfigurable resources. The architecture offered is generic, so that it can facilitate the deployment of any HW accelerator in any vendor's FPGA and FPGA family. An engineer can easily compile a deployable HW accelerator, while in turn multiple instances of the implemented accelerator can be spread around the multi-FPGA platform, offering the demanded parallelization. In this thesis a prototype system was designed and built, dealing with all the architecture's entailed complexities, and proving that UNILOGIC is feasible. Furthermore the complete system is evaluated, based on real world applications, accelerated on reconfigurable hardware and proving that UNILOGIC is scalable, while also being efficient both in terms of performance and power consumption.

1.3 Contribution

The contribution of this work comes from the proposed, implemented and evaluated UNILOGIC architecture. UNILOGIC offers an answer to the challenge that rises when attempting the utilization of multiple reconfigurable devices in an efficient and transparent manner. It offers users the ability to access the available hardware resources, and utilize/implement custom or popular accelerator modules. These modules execute an application's computationally-intensive algorithmic tasks concurrently, therefore, taking full advantage of the inherent parallelism of the HPC applications. In addition, the presented architecture allows for the accelerator modules to be implemented, i.e. placed or, in FPGA terms, configured close to where data are stored or, alternatively, move data into the memory located near the accelerator module that will processes them.

The key features of UNILOGIC are (i) direct access of the entire system memory by the accelerators using the PGAS model, and (ii) transparent use of all available FPGA resource in the system by one or multiple applications. The hardware architecture utilizes multi-core CPUs tightly integrated with the reconfigurable modules. A static part in the FPGA implements the inter-FPGA communication, the task scheduling and partial bitstream propagation into the dynamic reconfiguration part. The dynamic part hosts the partial bitstreams for the HPC application accelerated tasks. These modules are not bound to a single application, but on the contrary, can be dynamically reconfigured and shared among many different HPC applications.

In terms of the UNILOGIC system software, it comprises of a) the low-level device drivers, i.e. middleware that supports software-hardware communication, as well as the low-level API for monitoring and mapping accelerator modules onto the reconfigurable resources, and b) the runtime system that manages system resources as well as distributes and orchestrates the processing of HPC application data. The runtime is hosted on a lightweight, custom, embedded Linux OS, and allows users to effortlessly interact with the system as well as introduce application(s) at user-space; in addition, it assesses the system resource status so that an application is partitioned and distributed efficiently, thereby, allowing for in-parallel processing of its sub-tasks in reduced time frames.

The proposed UNILOGIC architecture was originally presented in [79], and involves a high-level introduction of the architecture, that integrates multiple FPGAs in a multi-processor environment. Then I moved on to a more detailed design of this concept, i.e. a parallel environment, to which I analyzed all its basic components, and importantly present detailed evaluation results for the first UNILOGIC-based multi-FPGA server prototype. Specifically, the main contributions of this thesis are the following:

- **A scalable architecture for multi-FPGA platforms** - The UNILOGIC hierarchical architecture supports efficient use of hundreds of reconfigurable accelerators distributed among hundreds of FPGAs.
- **Improved programmability in a multi-FPGA platform** - The UNILOGIC architecture allows for the seamless exploitation of the reconfigurable resources of the multi-FPGA system.
- **Sharing of FPGA-based hardware resources** - A hardware virtualization mechanism that allows for sharing of the same hardware tasks, i.e. tasks implemented in reconfigurable accelerators, by different applications, as well as the deployment of many different hardware tasks by a single application.
- **Low latency communication** - A communication infrastructure, based on the Advanced eXtensible Interface (AXI) protocol [7], designed and implemented for fast low-latency intra- and inter-FPGA connectivity that provides a very efficient and distributed interconnection.
- **Implementation of the first 64-FPGA UNILOGIC platform** - The developed UNILOGIC prototype, supporting 64 tightly interconnected FPGAs in two 1U chassis, was used for the deployment and the evaluation of the UNILOGIC architecture by running two real-world applications.

1.4 Thesis Organization

The structure of this thesis is as follows:

- **Chapter 2:** We present a thorough survey on related work, demonstrating the state-of-the-art and comparatively approaching the basic characteristics, showcasing both differences as well as common grounds with our proposed approach. Both multi-FPGA systems as well as smaller-grade systems that present interesting attributes are included, while we also consider communication infrastructure solutions as well as higher-level software tool flows proposed. Furthermore we comment on related commercial infrastructure, and finally report on linked work through collaborated projects.

- **Chapter 3:** We describe the proposed UNILOGIC architecture, that unifies the reconfigurable logic of numerous HPC nodes in an extendable, scalable, energy-efficient system. It is thus presented to the developer as a seemingly undivided, continuous entity.

- **Chapter 4:** We present the first implementation of the UNILOGIC architecture, reaching up to a fully functional prototype. Many innovative underlying hardware components and respective customizations are showcased, surpassing obstacles encountered during the process. We also present the firmware that complements the hardware components so as to compose a complete UNILOGIC system.

- **Chapter 5:** We initially focus on the optimization of a single-FPGA implementation, explaining quantitatively the performance bottlenecks and the improvements. Next, we optimize the UNILOGIC implementation in multiple FPGAs, optimizing the inter-FPGA efficiency so as to increase the overall application performance.

- **Chapter 6:** We evaluate the UNILOGIC architecture on the QFDB-based prototype. We progressively deploy and evaluate the architecture, introducing the hardware accelerators initially on a single FPGA and eventually on all the FPGAs of our prototype. Two opposite cases of algorithms, and thus two resulting diverse accelerators, are reported, a compute-intensive and a data-intensive, providing the means for a comprehensive UNILOGIC evaluation.

- **Chapter 7:** We provide directions for future work and possible extensions to our work, based on latest technologies and devices launched, together with current and future trends.

- **Chapter 8:** We conclude this thesis.

We would like to note at this point that the writer of this thesis aimed at producing text with a good flow throughout all chapters, so that it composes a concrete "story". This way, the *reader of this thesis* can find interesting information in any of the chapters/sections. However, it is also important to mention that effort has been also invested so as to, without interrupting the flow of this story-telling, separate the more detailed information on architectural and implementation related aspects, under sub-sub-headings. That is, under quad-numbered sections of the form x.x.x.x, e.g. 4.2.5.2. Based on this reasoning, a reader that wants to avoid getting into all the detailed techniques and entailed architectural approaches and solutions contributed by this thesis, and wants to gain an insight of a higher level for the information presented, can decide to selectively focus on these parts.

1.5 Details on Contributions and Collaborative Efforts

This thesis is part of the ECOSCALE project, which in parallel collaborated with sister projects ExaNeSt and ExaNoDe. Hence it entails a lot of collaboration and broader efforts by a number of participants, that made the realization of this result possible. Details on the contributions derived from this thesis are presented in the introductory paragraphs of chapters and their main sections, however we also briefly outline these here for ease of access. The main overall contribution, as already discussed, is the particularization of the UNILOGIC architecture stemming from [79], and its realization and evaluation on an effective hardware platform. The core architecture is formed through its "Worker" building block, which provisions all the main attributes of UNILOGIC. The Worker structure and all related intricacies are thoroughly explained throughout this thesis, while in Figure 1.1 a color-coded, high-level version of its block diagram aims to gives an insight on the specific contributions of this Thesis.

To be more specific and by referring to building blocks of the architecture, the author of this thesis:

◦ devised the topology exploration and formalization

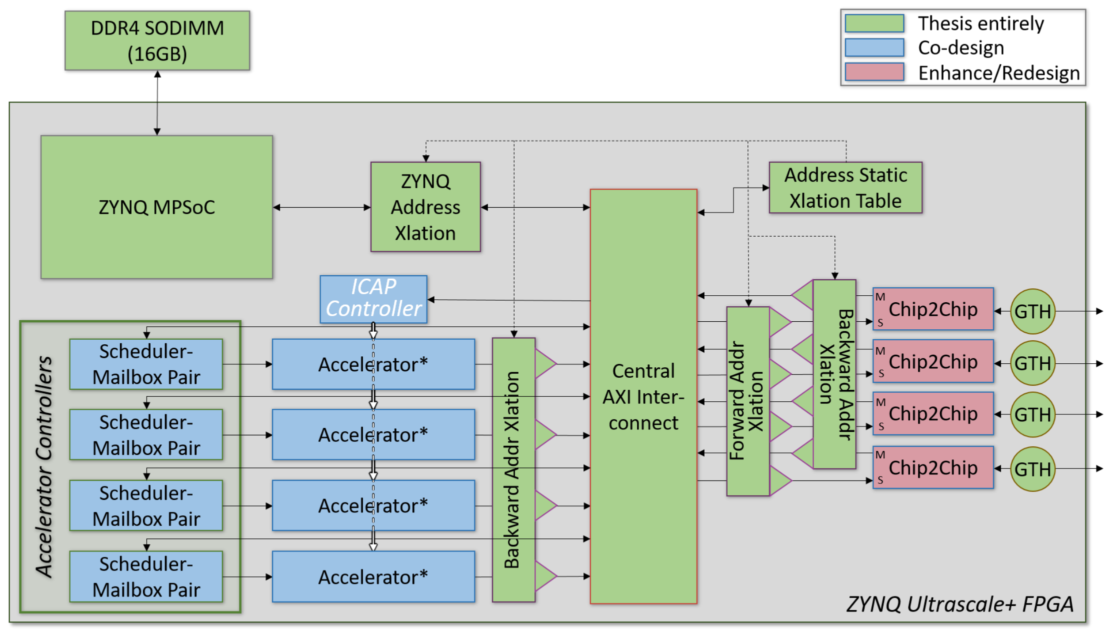


FIGURE 1.1: The architecture of a Worker, i.e. the main building block of UNILOGIC, shaded accordingly to clarify contributions of this thesis

- proposed and implemented the forward/backward Address Translation Scheme
- devised the tailored PGAS model
- designed the hardware translation tables, the algorithms and the related software for translation table configuration
- performed the initial MPSoC deployment
- took care of the QFDB-specific MPSoC & DDR configuration and fine-tuning
- analyzed connectivity through extensive multi-gigabit link bit error rate testing (BERT)
- devised the MPSoC PS-to-PL Addressing Scheme
- designed and built the whole memory port virtualization scheme, as well as the related investigation on optimized selection of port sets and exploration on favorable path to port designation
- designed the selectively hierarchical AXI interconnect that achieves highly reduced FPGA resources, as well as the related optimized AXI address mapping
- devised the thorough AXI parameterization, including pending transactions, registering and selective mapping, as well as Round Trip Time vs. transactions formalization
- explored the scheduling-accelerator slot pairing & thoroughly evaluated alternatives

- contributed in the Scheduler specs, followed by co-design mainly by Chalmers University, and then I performed the full hardware deployment and most of the related software
- contributed in the QFDB design mainly done by FORTH, in bring Up process, while I solely deployed it in the UNILOGIC platform
- fully designed and realized the RAM-less design and bare metal testing, with the Linux OS mainly devised by FORTH.
- highly contributed in the baseboard design while devised most of the the bring Up and deployment process
- fully carried out the Final hardware platform realization as well as its remote management infrastructure, while partly contributed in the related Linux OS and Drivers
- updated and enhanced the Conv-ID module adopted by UNIMEM
- devised ample investigation on the chip-to-chip module adopted by UNIMEM, including debug, enhancements, support for the GTH transceivers, designed clock-slave/clock-master versions, built 5, 10 and 16 Gbps versions, and contributed in the deployment of the custom transceiver protocol as well as for link bonding support
- provided specifications for the partial reconfiguration (PR) slots, the PR slot merging and the partial bitstreams, which otherwise are mainly a contribution of the University of Manchester
- designed and realized the PR-slot surrounding infrastructure
- persistently devised and executed bare metal testing and optimization
- contributed in the platform's Linux OS & drivers which otherwise was mainly contributed by TSI in Chania
- slightly contributed in the accelerator algorithms (Michelsen, Hyperbolic, KNP) & HLS which otherwise was mainly contributed by Synelixis, PoliTo and Acciona
- Highly contributed in the co-design process of accelerator optimization and evaluation, as well as the exploration on algorithmic level (e.g. Michelsen vs. Hyperbolic) & and on the level of optimization
- mainly devised the power measurements & the related automation scripting
- slightly contributed on the software measurements for CPUs/GPUs, otherwise mainly contributed by Synelixis, PoliTo and Acciona.

Chapter 2

Related Work

In this chapter we will visit some of the main, state-of-the-art reported work that closely relates to this thesis. We will first report related work on FPGAs directly related to HPC, proving the efficiency of FPGAs as compared to CPUs and GPUs, however mostly pertaining to single FPGA implementations, and emphasizing on the low FPGA programmability drawback. We will then visit multi-FPGA systems, where FPGAs are mainly used as coprocessors, hosted on boards and connected through PCI or Ethernet, however with very limited direct inter-FPGA connectivity. Work on communication infrastructure is then reported, pointing out the importance of custom communication infrastructures for the overall performance, and the potential of the FPGA alternative to offer such, whereas only small scale evaluations are reported. Then, in the resource virtualization field, interesting work is presented on single FPGA resource virtualization, as well capable frameworks and related middleware for effective multi-FPGA resource management. We also visit commercial multi-FPGA infrastructures, where custom solutions are built to target specific workloads and achieve performance while economizing on power consumption. An overview of the road map to exascale is also reported. Finally our work is identified as part of leading and closely collaborating EU HPC projects.

FPGAs in HPC:

In the last few years, there has been an increasing interest in employing FPGAs in the HPC domain. Escobar et al. in [23] present a thorough survey on implementing algorithms in heterogeneous HPC infrastructures that integrate diverse resources, including CPUs, GPUs and FPGAs, and provide guidelines for effectively employing FPGAs in HPC. In this survey, it has been estimated that half the lifetime cost of HPC platforms is devoted to electrical power. Moreover, another important observation in this survey is that, in order to overcome the low-programmability issues of older FPGA-based architectures, high level synthesis tools such as Vivado HLS (by Xilinx) and Catapult C (by Mentor Graphics) have been recently heavily improved. It is also noted that a basic part of the success of CPUs and GPUs, in the HPC domain, is due to the widespread adoption of libraries, in contrast to the existing custom FPGA based solutions. This thesis addresses the majority of the main issues raised by this survey paper, since our work fundamentally focuses on improving the programmability of multi-FPGA environments within HPC infrastructures, while, in parallel, we provide a low latency communication architecture, scalable to a very large volume of interconnected FPGAs.

There are several other researchers [12, 15, 64, 17, 78, 80, 83] presenting encouraging performance results for reconfigurable systems, especially when comparing the performance per watt of the FPGAs against that of their CPU and GPU counterparts. Moreover, these papers also emphasize the numerous programmability issues when designers build FPGA accelerators using common FPGA design tools. Crucially, the work presented here stands apart from all those papers because their focus is on the utilization of a single FPGA.

Multi-FPGA systems:

In the last decade, there has been an increasing interest in multi-FPGA systems, such as the LEAP system [30], the JetStream [116], the CUBE Cluster [125], the Melia framework [121], the FPGA Groups [52] and the Blaze FPGA Accelerator [42]. Also, there are the Novo-G# system [33], which integrates the 192 Stratix IV, the Amazon EC2 F1 instances containing up to eight FPGAs [44], the Maxeler MPC nodes [47], which accelerate applications on reconfigurable Data Flow Engines (DFEs) [87, 88, 46], and the Rivyera architecture from SciEngines supporting up to 128 Xilinx Spartan FPGAs per machine [35]. Most of these systems employ commercial FPGA boards, such as the ones developed by Bittware [43, 11], Hitech Global [71] and Digilent [45], and rely on Intel processors connected to the FPGA boards over PCIe, however, such an interconnection scheme entails high communication latency in the order of tens of microseconds, usually introduces FPGAs only at the edges of large computation systems, while it also usually entails increased aggregate power consumption.

The Convey with HC-2 [18], composes a hybrid-core computer that tightly integrate two industry standard Intel Xeon processors with four Xilinx Virtex FPGAs as corpocessors, along with a very powerful memory subsystem. Also, there are approaches that provide tightly coupled reconfigurable resources, such as BEE7 from BEEcube (now acquired by National Instruments) [9, 50] and SG280 from ProFPGA [99, 34]. IBM proposes a multi-FPGA platform [122] that targets the Data Center (CloudFPGA), and uses ethernet connectivity while mainly focusing on decoupling the FPGAs from the CPU. Ethernet however results in a less efficient interconnect, at least in terms of latency [16, 41, 117]. The CloudFPGA does not aim to unify the FPGA resources as our architecture does, however this work also advocates that the FPGA-to-CPU PCI connectivity should be relinquished and effort should placed at providing architectures for self-contained FPGAs. In essence, it becomes evident that in order to use such multi-FPGA infrastructures within an HPC system, it is crucial to employ a highly parallel architecture, which will offer ease of hardware resource utilization to the accompanying tool/programming flow, and will meet the HPC requirements, e.g. low communication latency, such as the UNILOGIC approach.

In [72], a multi-FPGA platform is built and evaluated, constructing a many-core hardware prototype and focusing on parallel programming research. 64 single-FPGA "Formic" custom boards hosting a total of 512 MicroBlaze processor cores, along with 8 ARM A9 processors, that constitute a 520-core heterogeneous prototype, which being greatly faster than simulators, can elevate parallel programming research. The multi-FPGA approach of [70] is proposed as Network-on-Chip (NoC) emulation Framework. A large sized five-FPGA board is used to host processing cores and network elements, greatly outperforming software simulators. 32-bit RISC processors configured in the FPGA, along with switching elements, are emulating 2x2 and 4x4 NoCs in various configurations. The AMBA bus is used for communication inside the FPGAs, and the multi-gigabit transceivers along with on-board parallel wires are deployed for cross-FPGA interconnection. However, this approach does not target the accelerator field, while during the evaluation process, cross-FPGA connectivity is not stressfully tested, as only low communication-demanding applications are demonstrated.

A study presented in [107, 108] focuses on an efficient design flow to create custom, FPGA-based, prototyping platforms, while taking into account a description of the target design, which may include a single or multiple interconnected boards. A comparison among alternatives ranging from off-the-shelf to custom solutions is also presented. In [66] the authors present the TaPaSCo framework, that aims to automate the construction of System-on-Chip (SoC) FPGA designs for task parallel computation. In the case study presented, an FPGA is deployed as a coprocessor next to a CPU, and uses MicroBlaze soft-core processors to offload host CPU tasks.

Communication infrastructure:

Alongside and regardless of the CPUs' processing speed, major bottlenecks in high performance architectures are triggered by the vast data transfers, the slow memory hierarchy and the relatively high inter-node communication latency. The reconfigurable nature of FPGAs allows for implementations of custom efficient communication infrastructures which are crucial for the overall system performance as explained by Correa et al. [20] and Viswanathan et.al. [118].

In [20], the authors propose a scalable multi-FPGA interconnection architecture implemented and evaluated on a small prototype comprising of 8 FPGAs. Similarly, a small prototype is used in [118]. On the contrary, we have implemented and evaluated our proposed communication infrastructure on a larger prototype consisting of 64 FPGAs, demonstrating with more confidence the scalability and the effectiveness of the proposed solution. Mondigo et.al. [81] propose a multi-FPGA communication infrastructure that uses single FPGAs as pipeline stages. This architecture can be very efficient only for certain applications.

Furthermore, interconnection topologies are studied in [60], describing the pros and cons for different ones such as the fully-connected and torus. Here, we present an enhanced torus-like topology that combines the pros of both worlds. Moreover, Ethernet is commonly used in several FPGA infrastructures [109, 122]. Our approach supports both standard Ethernet connectivity for inter-processor communication and custom FPGA-to-FPGA high-bandwidth and low-latency interconnection for highly-demanding hardware acceleration.

Finally, Kapre et al. [59] focus on low-level intra-FPGA communication, implementing a custom FPGA Network-on-Chip (NoC) router that can be faster than the overlay routers available today, while achieving an order of magnitude reduced size. In our prototype we are using the Xilinx AXI interconnect but our architecture could employ other routers as well.

Resource Virtualization & Frameworks:

Research in FPGA virtualization is still at a very young age. RACOS is a Reconfigurable ACcelerator OS (RACOS) that supports the concurrent use of multiple accelerators by multiple applications by providing a software API to load/unload reconfigurable hardware accelerators and share them between multiple processes in a single-node system with PCIe-attached FPGA board(s) [115]. The VINEYARD approach [56, 55] offers resource virtualization which supports openCL based accelerators, while in parallel proposes a high-level framework for their efficient utilization in the data centres. This kind of frameworks [54], as well as relative commercial schemes such as the Coral [53] offered by InAccel, could take advantage of the UNILOGIC architecture, complementary offering a holistic approach for FPGA deployment. The work in [104], addresses the virtualization of hardware accelerators through the Single-Root I/O Virtualization feature of the PCI Express interface. The proposed system is capable of statically sharing predefined co-processors in a single FPGA among a host and several virtual machines; therefore, co-processors are not shareable between domains, while in [120] the system gets augmented with partial reconfiguration support. In [3] A. Al-Aghbari et al. present a single FPGA virtualization approach targeted at clouds and data centers. Vaishnav et. al. [114] highlight in their survey the importance of reconfigurable resources virtualization at three different levels: the accelerator level, i.e. inside an FPGA, the node level (few FPGAs) and the system-wide multi-node level. A few solutions per level are referenced in this survey. Our proposed architecture is probably the only one which efficiently targets all three designated levels, giving maximum flexibility along with the desired transparency that allows the programmer to be fully unaware of the underlying virtualization mechanism.

FPGA-based parallel commercial infrastructure:

There are several commercial parallel systems that demonstrate that FPGAs constitute an energy efficient and flexible choice for several parallel applications. Microsoft uses FPGAs in its Bing search engine [98, 97] under the Catapult project to achieve 95% higher performance at the cost of 10% higher power consumption. Cray delivers a CS500 cluster system with Stratix FPGAs for the Noctua project [93]. Baidu is using low cost FPGAs to accelerate Deep Neural Networks [85], while IBM deploys FPGAs for large NoSQL data stores [13].

Road map to exascale:

Finally it is important to report on the significance of the exascale era and the related efforts to realize exascale computing. [29] highlights the continuous scale up in terms of nodes and accelerators, as well as software, infrastructure and tools on the road to exascale. In the European Union side, [105] stresses that European supercomputing infrastructures represent a strategic resource for European citizens in the years to come, as well as for the future of European industry, small and medium-size enterprises (SMEs), and the creation of new jobs. The U.S. exascale computing strategy is described in [67], presenting the vision of an exascale ecosystem adjoining capable and power efficient computing platforms, with enhanced applications, software, and hardware technologies. Nowadays, being really close to the realization of such systems, [101, 27] report on exascale computers that are just around the corner.

Linked work:

The work of this thesis is part of the ECOSCALE project which collaborated closely with the ExaNeSt [62] and ExaNoDe [100, 10] EU projects, as successor projects to the Euroserver project [22]. ExaNeSt main focus was on the interconnection infrastructure, while ExaNoDe main focus was on improving the on-chip packaging technologies. ExaNeSt has designed and implemented the QFDB compute node that we use as a building block in our final prototype (see Section 4.1). Finally the currently ongoing EuroEXA project [24], inherits the findings of all these projects.

Chapter 3

The UNILOGIC Architecture

In this section we describe the UNILOGIC architecture [79] focusing also on the communication infrastructure of the HPC platform, which allows for seamless and fast access of any data and any computational logic resource in the system. The target of the UNILOGIC architecture is to provide an extendable, scalable, energy-efficient system, unifying the reconfigurable logic of numerous HPC nodes, and presenting it to the developer as a seemingly undivided, continuous entity. In this thesis, the originally proposed generic UNILOGIC architecture [79] gets concretized in it complete form as presented herein. Also, the whole communication infrastructure was devised as part of this thesis.

3.1 Description of UNILOGIC

The UNILOGIC approach is an extension to the UNIMEM (Unified Memory) architecture [77]. UNIMEM has been a novelty in its own right and its purpose is to provide a uniform memory address space across multiple HPC nodes. It was first introduced within the EuroServer project [25], [22] and it consists of a powerful set of mechanisms that provide efficient communication among remote CPU-based nodes of a large HPC system. The main advantage of the UNIMEM architecture over conventional communication architectures, i.e. coherent shared memory systems and message passing computational systems, is that it offers more efficient communication mechanisms than the conventional message passing systems and eliminates the complexities, performance overheads, and costs that the large coherent shared memory systems induce.

An HPC system implementing the UNIMEM architecture consists of a set of computational nodes that are connected through a custom network. UNIMEM enables the nodes to seamlessly access data items located in remote nodes. More specifically, in the UNIMEM architecture, the memories of the system are mapped into a Partitioned Global Address Space (PGAS) that is accessible by any node. Therefore, any node in the system can directly access the physical memory of any other node through the GAS.

In the PGAS of a UNIMEM multi-node machine, a memory page can be cacheable at the local processing node or at a remote processing node, but not at both, as seen in Figure 3.1. This is the basis of the UNIMEM consistency model, which eliminates global-scope cache coherence protocols, providing a scalable solution. Progressive address translation [61] can be further applied on top of UNIMEM in order to provide interprocessor communication.

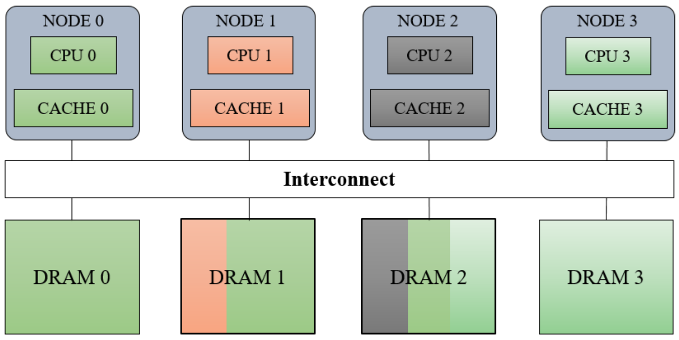


FIGURE 3.1: Unimem allows pages of DRAM1 cached in CACHE0 of CPU0 -OR- in CACHE1 of CPU1 etc.

UNIMEM allows remote DRAM borrowing and remote load/store instructions, which enable remote-mailbox and remote-interrupt notifications for low-latency protocols. It also allows communication using Remote Direct Memory Access (RDMA) operations, which efficiently deliver data in-place and avoid receiver-side copying. The complexity and costs that the system-level coherence protocols induce [69] are eliminated in the UNIMEM architecture, as it imposes exclusive caching.

Subsequently, UNILOGIC has been envisaged as an extension to UNIMEM, and introduces the uniform and virtualized access of reconfigurable logic, i.e. of acceleration resources, residing in the different FPGA-populated nodes of the HPC heterogeneous system. Since a fundamental aim is the extension to a very large number of heterogeneous nodes, which incorporate FPGAs, so as to eventually reach exascale capabilities, the UNILOGIC architecture has been developed to be inherently scalable. The way UNILOGIC expands upon UNIMEM is by including *accelerator controllers* in the UNIMEM architecture that can be seamlessly accessed by any node in the system in order to accelerate tasks in the reconfigurable resources. Furthermore, partial reconfiguration can be employed in order to load new hardware-accelerated tasks in the reconfigurable resources or to relocate a task to any node in the system [65].

The UNILOGIC architecture partitions the system design into several processing nodes, called Workers, which communicate through a hierarchical communication infrastructure or mesh-like topology. Each Worker comprises of conventional processing units, memory, reconfigurable logic, and accelerator controllers that provide access to the reconfigurable resources in order to program and execute accelerated tasks in hardware at runtime (Figure 3.2).

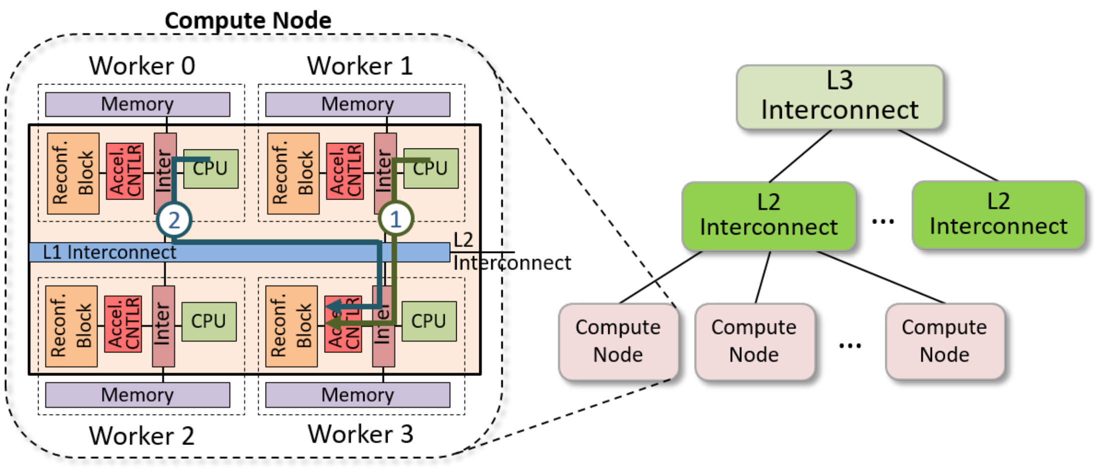


FIGURE 3.2: The Hardware Architecture

The coding framework used has been that of OpenCL [110, 84] since an OpenCL kernel call is split into OpenCL *Work Groups* (WGs), which can provide coarse-grain parallelism among different FPGAs. Subsequently, each WG is further split into OpenCL *Work Items*, which can provide fine-grain parallelism within an FPGA.

The main advantages of UNILOGIC are the following: (1) the architecture includes accelerator controllers, which can be reached using memory accesses that are part of the UNIMEM PGAS, making the controllers visible by any system node, (2) accelerated tasks in the reconfigurable logic can access directly any data in the UNIMEM PGAS via regular data transactions or via initiating block transfers in an RDMA fashion, and, finally, (3) resource sharing is supported by the accelerator controllers by serving requests from different nodes in-parallel, as depicted in Figure 3.2 where two nodes access controllers of a third, remote node.

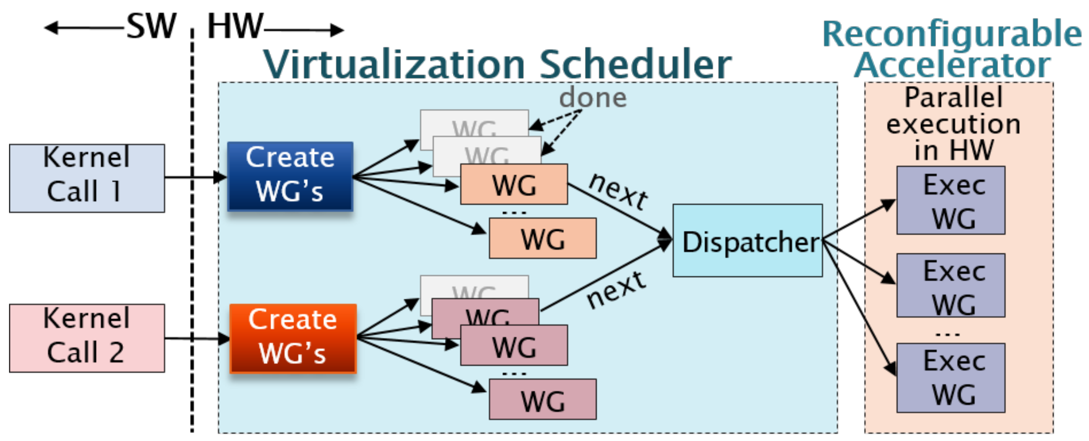


FIGURE 3.3: OpenCL Kernel WG Distribution with the Virtualization Scheduler

The resource sharing requires the inclusion of a *Virtualization Scheduler*, which allows for the execution of multiple requests for the same function to be executed in-parallel. As shown in Figure 3.3, upon the execution of an OpenCL kernel call, the Virtualization Scheduler creates the WGs of the kernel, which are dispatched for execution in hardware. Since the hardware reconfigurable resources are limited, a small number of WGs can be implemented in hardware and executed in parallel. The *Reconfigurable Hardware Accelerator* includes a number of the same WG implementations, i.e. a number of identical parallel hardware pipelines that can operate in a superscalar fashion, which actually corresponds to the number of the outstanding WGs that can be scheduled in-parallel by the Virtualization Scheduler. The Virtualization Scheduler needs to remember only what WGs have been executed or scheduled for execution so far. Furthermore, it can mix the execution of the WGs from different calls for the same OpenCL kernel, as well as provide Quality of Service (QoS) by controlling the rate at which the WGs are executed.

In an HPC application, virtualization and context switching enables multiple tasks or threads to share a single CPU in order to maximize the utilization of the CPU resources. Similarly, our UNILOGIC architecture supports fine-grain sharing of the FPGA resources, where a function implemented in hardware, which can be “called”, i.e. invoked, by different tasks or threads running on different

HPC nodes, in parallel, through a custom-designed hardware Virtualization mechanism. This virtualization/scheduling block will allow *multiple function calls from different HPC nodes to be executed in a fully pipelined fashion.* This can be seen depicted in Figure 3.2, where both applications running in Worker 0 and Worker 1 can employ the reconfigurable accelerator of Worker 3. They do so by accessing the corresponding accelerator controller, which will then properly schedule and dispatch the tasks to the hardware WGs. The accelerator in turn, as directed by the application, may have to proceed by accessing data either from its local memory, or from the memory of any other Worker. A second case scenario, is that of an application accessing many accelerator controllers. In this case, tasks can be send over to accelerator controllers of either the same or different kinds, and either on the local or on many/remote Workers. This way we can allow for a single application to a) assign a large computational task to many identical hardware accelerators, by accessing controllers in a single or in any number of discrete Workers, b) assign different computational tasks to diverse accelerators, by accessing controllers in a single on any number of discrete Workers. More detailed figures and descriptions on openCL kernel execution and virtualization can be found in Appendix A

Moreover, the UNILOGIC architecture supports coarse-grain time-sharing of the reconfigurable resources through partial runtime reconfiguration. Partial reconfiguration is supported both locally and remotely, i.e. any node in the system can invoke partial reconfiguration to an FPGA in any other remote node. The partial reconfiguration mechanisms have been already integrated and evaluated in our platform, but are outside the scope of this thesis; the reader can find more details on this work in [90], [91]. However, as a lot of effort that was invested as part of this thesis in order to support the mechanisms developed by the University of Manchester, and to effectively integrate those in the UNILOGIC architecture, hence part of the solutions contributed will be described.

Depending on the UNILOGIC implementation, the functionality of the Virtualization Scheduler can be implemented in software, in hardware (inside the Acceleration Controllers) or both. A software application may send a large task to an accelerator controller, which can then in hardware split this large task into WGs, based on the configured hardware accelerator it controls. Otherwise, the software itself can choose to deal with the WG splitting, and likewise to properly dispatch the resulting WGs. Similarly, the more complex role of orchestrating concurrent, local and remote running applications, that ask to be serviced by the same underlying hardware, i.e. accelerators, can be relied upon hardware or software.

3.2 UNILOGIC Communication Infrastructure

The fundamental aim of UNILOGIC is to enable hardware modules, such as accelerators controllers and hardware WGs, located in any of the numerous Workers, to communicate through a unifying and easily configurable interconnect, while appearing within a single, consistent PGAS and seemingly presented as if they all reside within a single node. This leads to a completely unified and virtualized Worker environment, supporting intercommunication in a uniform, seemingly local-access manner. In that respect, it allows data transfer or, symmetrically, computation task migration, in such a way that the system will behave as if everything is executed within a single node, i.e. in a vast, contiguous and reconfigurable resource space. Details on the communication infrastructure implemented on the UNILOGIC prototype, are presented next in Section 4.

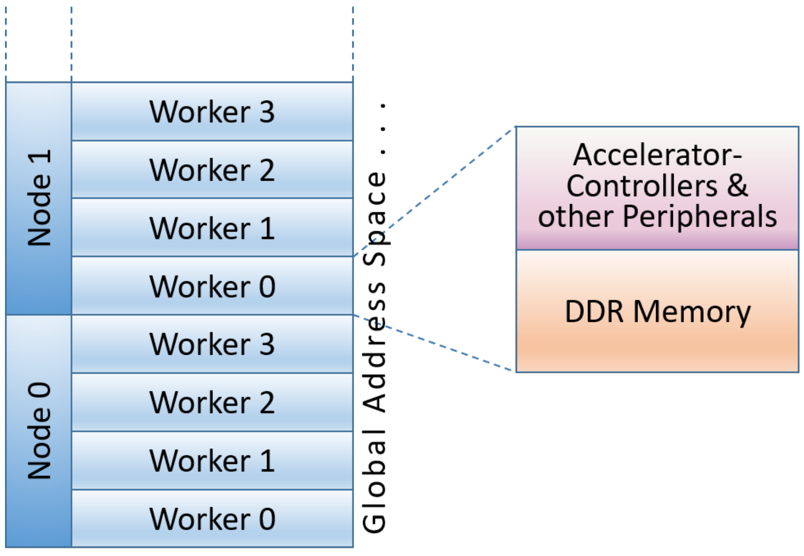
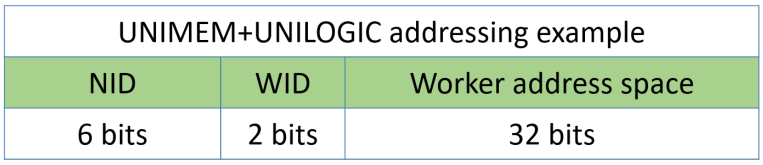


FIGURE 3.4: A generic view of the UNIMEM+UNILOGIC global address space. Each Worker corresponds to an address window

The novel UNILOGIC inter-communication infrastructure, developed so as to provide this multi-Worker memory and hardware resources unification, adopts the PGAS paradigm depicted in Figure 3.4. On the left, each Worker has its own dedicated and continuous address region in the PGAS. Workers are grouped into

Compute Nodes. Finally, many such nodes make up the complete PGAS. On the right, each Worker is shown with its own address space, comprising of a memory region and a region that corresponds to the accelerator controllers and other peripherals. The latter is used for accelerator invocation, peripheral configuration, local or remote partial reconfiguration, etc.



UNIMEM+UNILOGIC addressing example		
NID	WID	Worker address space
6 bits	2 bits	32 bits

FIGURE 3.5: An generic example for a global address resolution

An example address is shown in Figure 3.5 and comprises of three basic fields, the Compute Node ID (NID), the Worker ID (WID) and the remaining address bits that pertain to the address space within each Worker. The width of these fields is implementation-dependent and can grow as much as required for a system to scale. In the example depicted, the NID's 6 bits support up to 64 nodes, with each node integrating 4 Workers as implied by the WID's 2 bits. Therefore, this example address resolution is configured to support $64 \times 4 = 256$ Workers, with each Worker exposing 4 GBytes (32 bits) of address space, thus accumulating to a total global address space of 1 TByte ($6 + 2 + 32 = 40$ bits).

Chapter 4

UNILOGIC System Implementation

In this section we present the first implementation of the UNILOGIC architecture, reaching up to a fully functional prototype. We first present the innovative underlying hardware components that build up our custom prototype. For each component we highlight the implemented architectural aspects as well as the main implementation obstacles encountered and the corresponding solutions. We then present the firmware that complements the hardware components so as to compose the complete UNILOGIC architecture. Details on the author's exact contributions are given through the introductory paragraphs in the sections of this chapter.

4.1 Low Level Architecture: Hardware Components, Topology & Prototype

The UNILOGIC-based prototype comprises of custom built components that are designed and built in order to support efficient, ultra-high density, FPGA-based, compute-node oriented infrastructures. Moreover, many of the implemented features may be utilized in other parallel FPGA platforms. We first present the custom-made multi-FPGA board (called Quad-FPGA Daughter Board - QFDB) that achieves a very dense, tightly interconnected FPGA placement. We then present our approach on the interconnection topology, with the target to better exploit the potential of the QFDB's available outgoing links. We then move on to the next level of building blocks, i.e. the hosting boards for the QFDBs, and present our custom 1U server board that hosts eight QFDBs. Finally we gain insight on our final prototype, used to fine-tune and evaluate UNILOGIC.

I was highly involved in the QFDB board design presented herein, as well as the bring up process and subsequent deployment. The extreme memory borrowing scheme is also based on hw fully implemented by the author of this thesis, while the specialized sw stack was designed by specialized collaborators. I also devised the whole topology along with the corresponding formalization, while I took great part in the baseboard specifications, and took care of most of the subsequent deployment. Finally I set up the full prototype along with all the local and remote management automation, while I was greatly involved in the SW stack co-design, including the custom Linux OS and device drivers.

4.1.1 The Quad-FPGA Daughter Board

Our basic compute node is implemented through the custom-made Quad FPGA Daughter Board (QFDB) [14, 4], which, as its name also denotes, hosts four FPGAs. In particular, each QFDB supports four state-of-the-art Xilinx Zynq Ultrascale+ devices (part number: xczu9eg-ffvc900-2-i). Each such FPGA device contains a quad-core ARM A53 Processing System (PS), and significant reconfigurable resources referred to as Programming Logic (PL), tightly coupled with the PS, thus it is also referred to as a Multi-Processor System-on-Chip (MPSoC). In addition, the QFDB board couples 16 GB of DDR4 memory and a 64 MB QSPI memory to each FPGA.

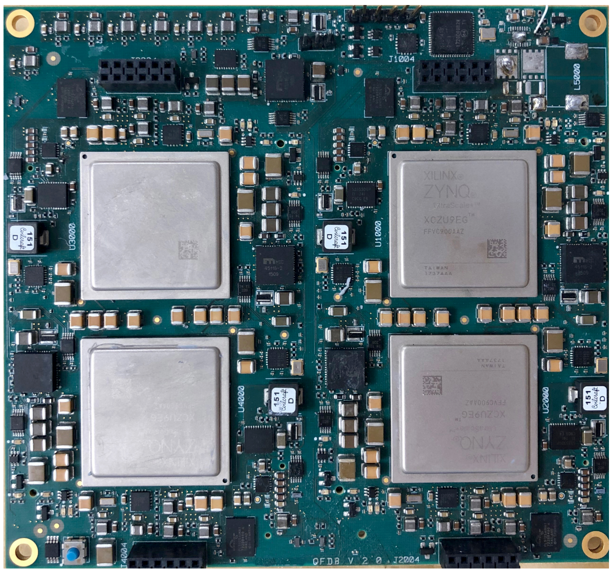


FIGURE 4.1: Top view of the QFDB

Targeting a compact design, the QFDB dimensions are squeezed down to the extremely compact 120×130mm, while no component on top or below the printed circuit board (PCB), seen in Figure 4.1, is taller than 10mm. The PCB stackup

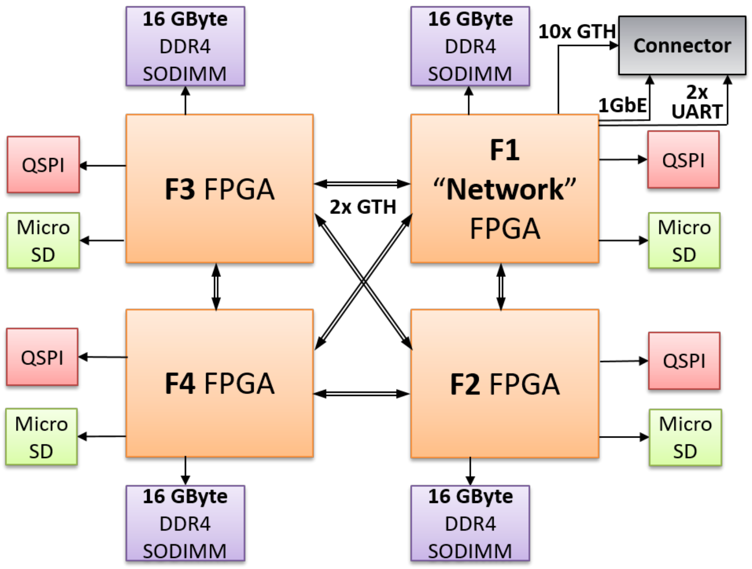


FIGURE 4.2: The QFDB block diagram

consists of 16 layers using Megtron-6 dielectric. The high concentration of components and high-speed traces required significant effort for the placement of the components and the routing of the traces.

There are also numerous interconnection paths, as shown in the simplified architecture presented in Figure 4.2, so as to allow for the efficient implementation of the UNILOGIC architecture. Each FPGA is connected to the outside world through 10 High Speed Serial Links (HSSL) by means of Multi-Gigabit Transceivers, referred to as "GTH Transceivers" by Xilinx. Each external link has a maximum line rate of 10.3125Gbit/s. A MAC-to-PHY RGMII interface also allows Gigabit Ethernet (GbE) connections which are mainly used for management purposes. Within the board, the FPGAs are connected together through 2 HSSLs operating at the maximum line rate of 16.375Gbit/s.

The Processing System (PS) of each FPGA device is configured with all four ARM A53 cores operating at their maximum 1333 MHz frequency. Each of the paired DDR4 memories (SODIMM) is ECC-protected and interfaces with the FPGA device through a 64-bit wide datapath. The DDR controller is configured to run at DDR4-PC2133, which actually exceeds Xilinx's MPSoC specifications for SODIMM modules, as the precise routing of the QFDB allows for higher speed than conventional SODIMM based boards. The coupled QSPI memory is used as a primary boot device. The reconfigurable logic features 274K LUTs, 2520 DSP cores and about 4 MBytes of SRAM memory (Block-RAM).

The QFDB also incorporates a number of I2C-compliant sensors, including both power sensors that measure current and voltage, as well as temperature sensors. Those sensors have been utilized in the evaluation of our system so as to have the most accurate results possible. In our initial prototype, we mounted QFDBs on top of unit carriers called Mini-feeders. Each such carrier hosts a single QFDB, so a corresponding prototype is build of QFDB + mini-feeder pairs. However, in our latest prototype, we utilized our custom and powerful carrier board, which can host eight QFDBs, and is presented below in section 4.1.4.

4.1.1.1 An extreme case of Memory Borrowing, devised to expedite QFDB bring up: RAM-less Linux boot

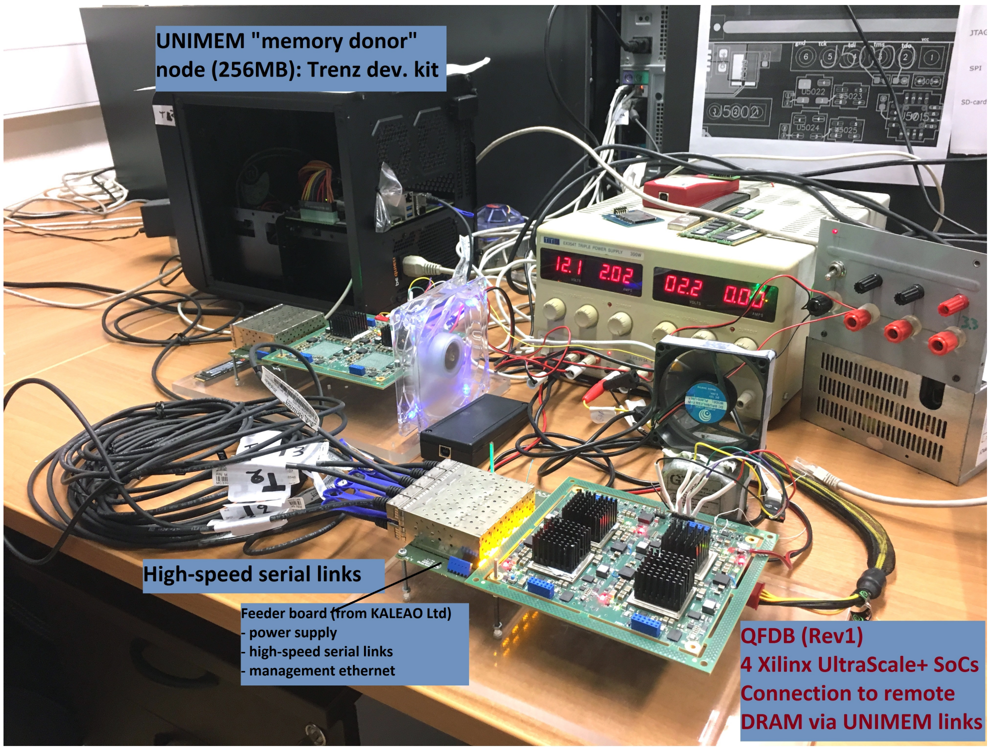

UNIMEM "memory donor" node (256MB): Trenz dev. kit
High-speed serial links
Feeder board (from KALEAO Ltd)
- power supply
- high-speed serial links
- management ethernet
QFDB (Rev1)
4 Xilinx UltraScale+ SoCs
Connection to remote DRAM via UNIMEM links

FIGURE 4.3: The prototype implementing the extreme case of memory borrowing: RAM-less boot

At this point it is useful to present an important step at the evolution of the QFDB based prototype, as it proved quite beneficial for both the UNILOGIC implementation, and for other platforms deploying this module, like the ExaNeSt project's liquid cooled prototype. During the bring-up of the QFDB, a critical error in the DDR connection was uncovered, which rendered DDR unusable, and thus hindered OS booting. At this point, a redesign and remanufacture (respin) of the board was needed, and this process would render a lot of time unproductive in respect to QFDB bring up and exploitation. This is so as, while prohibiting Linux boot, testing other peripherals on the board is likewise restricted, as is the case for Ethernet connectivity or SD card communication. Importantly, testing all components before conducting a respin is critical, in order to incorporate all needed modifications in a single reproduction cycle.

Nevertheless, we seized the chance to investigate on a memory-borrowing environment, which aims to orchestrate diverse components, and allow Linux boot under complete absence of local memory. This way we it would allow us to speed up the validation of the board. In this setup, presented in Figure 4.3 and explained through the block diagram of Figure 4.4, the Linux OS was booted by borrowing the memory of a third-party "donor" board. In our case, a Trenz TEBF0808 [36] was used as a memory donor, which features an identical Zynq Ultrascale+ FPGA along with 2 GB of DDR4-RAM.

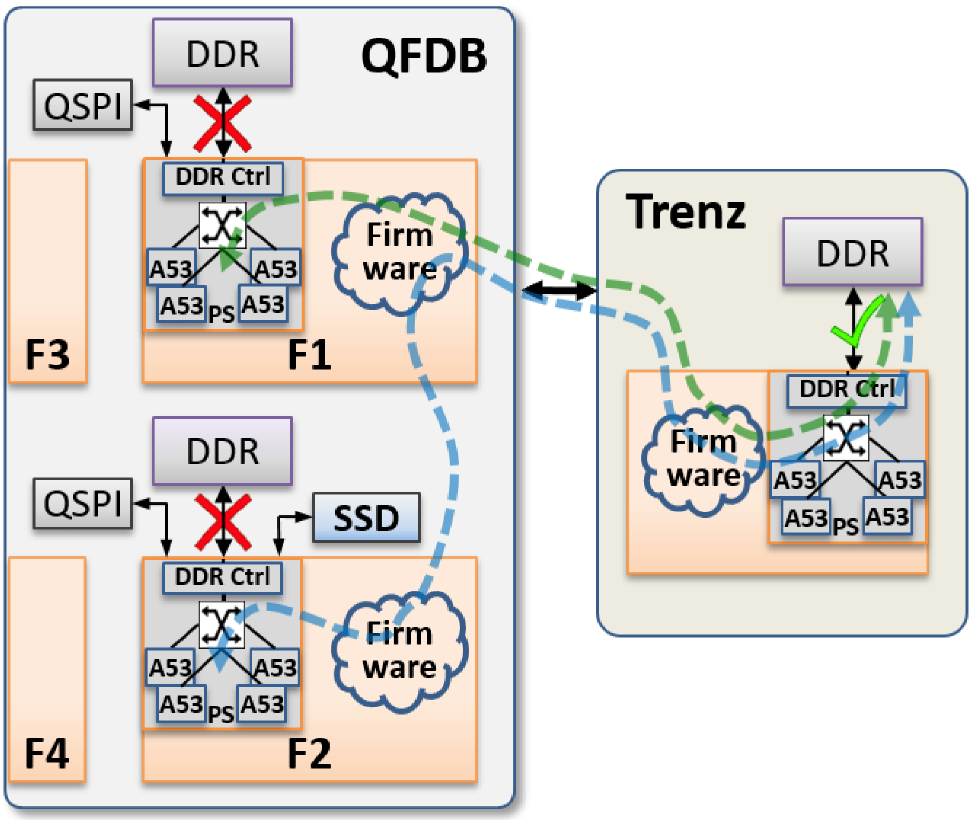


FIGURE 4.4: The block diagram of the HW design for the RAM-less OS boot through memory borrowing

As indicated with arrows, the F1 FPGA, unable to access its coupled, external DDR memory, and likewise no other DDR memory on the QFDB, has to divert memory access. All memory accesses of the processing system, are directed to the FPGA logic instead of the DDR controller. "Firmware" labeled clouds on the FPGAs, represent custom logic we specifically designed for this setup. This technique of exclusive memory borrowing may be applied to any other FPGA-based board lacking significant external memory.

However, not just using remote memory, but instead booting a Linux in this setup, presented many pitfalls. In effect, Zynqs are not meant to access program memory through the Programmable Logic. On the software side, a very minimal software environment was squeezed into the 32 MB of the QSPI flash memory. To that end, the U-Boot loader was trimmed aggressively, a custom Linux kernel (v4.9) configuration was used, and a minimal BusyBox setup was instantiated in the form of an initramfs image. The overall storage space used per node reached a mere 20 MB, including a 16 MB compressed bitstream. The First Stage Boot Loader (FSBL) was also modified not to require external RAM, and the code re-location of the U-Boot loader was bypassed.

On the firmware side, complementary designs were built on the QFDB and the Trenz board. Following Figure 4.4, any memory access from the QFDB MPSoC is steered to the Programmable Logic (PL), instead of reaching the local DDR-RAM. The memory access then exits the FPGA through a GTH transceiver. When reaching the other end, passing over SFP cables, the access is transformed using a specifically designed logic block, and reaches the functioning remote DDR through the Processing System (PS). The response then travels back in a similar manner. The whole mechanism is transparent to the software execution environment. The transceivers operate at their maximum lane rate (10.3125 Gb/s), which offers a reasonable memory throughput for moderate workloads. Each DDR-lacking node may just assume it uses a memory with higher latency.

Since only the Network FPGA of the QFDB is directly connected to the outside world, as seen in Figure 4.2, additional effort was required for the boot of the other FPGAs. Hence, the bitstream of the Network FPGA is enhanced to also act as a proxy, which forwards transactions for the other three nodes, and steers them to discrete memory ranges on the other end, as shown in Figure 4.4. 256 MB of memory where dedicated per FPGA, completely utilizing the 2 GB available on the Trenz "donor" board. With this approach, the F2 "Storage" FPGA was also successfully booted, with the Linux accessing its physical memory through the

F1 intermediate hop. This helped to earlier test, among others, the SSD (M.2) functionality, as the SSD is attached to the F2 FPGA of a QFDB. After having the SSD properly enumerated, and performed large I/O transfers we executed some early storage benchmarking.

4.1.2 Communication Infrastructure Topology

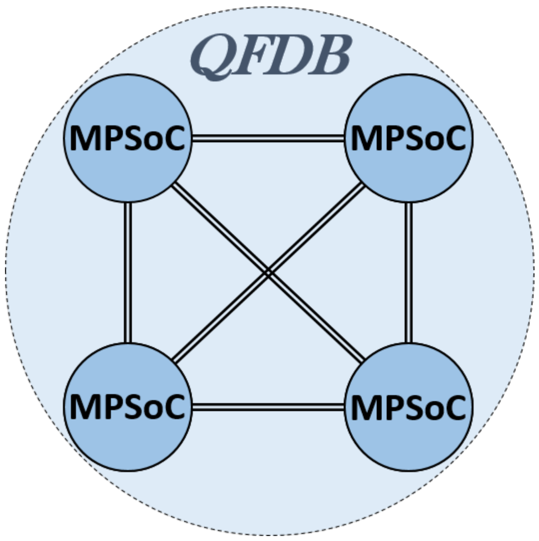


FIGURE 4.5: Intra-QFDB MPSoC connectivity

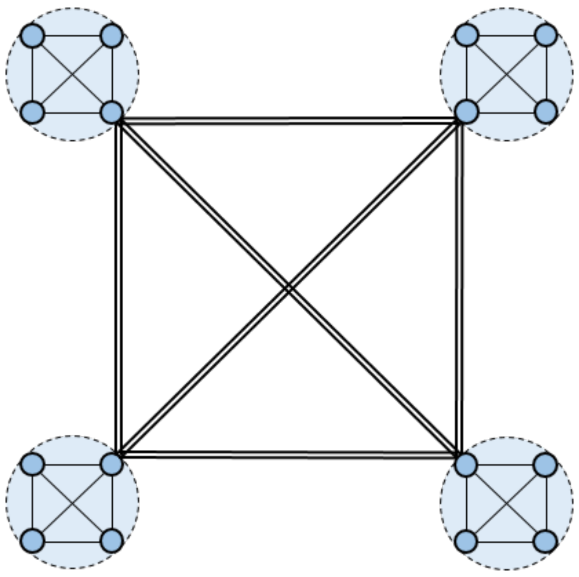


FIGURE 4.6: A quad of QFDBs fully interconnected

In terms of the interconnection topology, this derives partly from the results of the ExaNeSt project [62], as one of its prime efforts was to investigate the different interconnection schemes and propose the most efficient ones. In our case, we were driven by the need to increase the actual hardware interconnection density at its highest possible level since UNILOGIC's main feature is the efficient unification of reconfigurable hardware resources. Thus, since we deployed the QFDB as the basic building block of our implementation, we opted for exploiting its offered connectivity to the maximum.

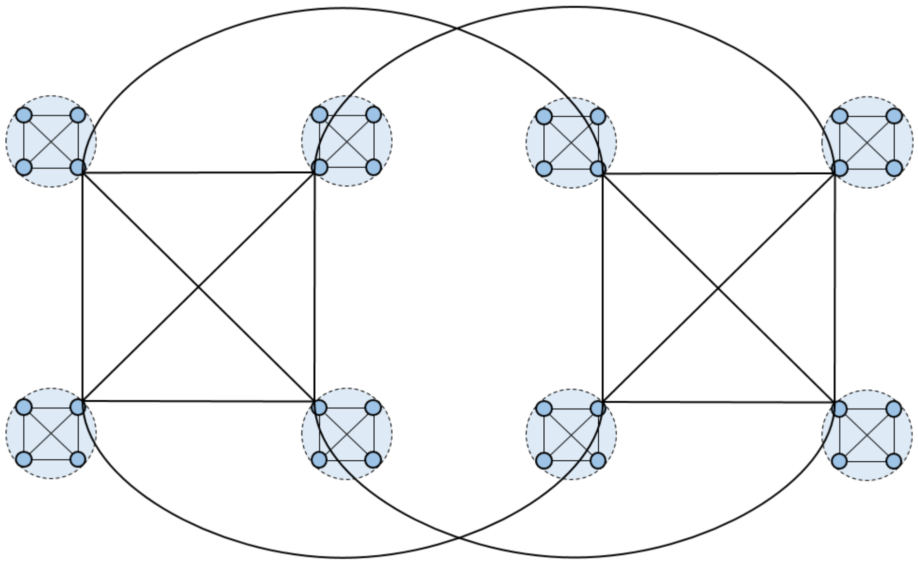


FIGURE 4.7: Unfolded Cube, as realized on UNILOGIC prototype's building blocks

As already presented in Figure 4.2, the QFDB offers an all to all interconnection topology (fully connected mesh) to its four hosted MPSoCs. Each such connection is realized through not a single, but a pair of links, i.e. GTH multi-gigabit transceivers. This intra-QFDB topology is abstractly presented in Figure 4.5. Then, moving on to inter-QFDB connectivity, each QFDB offers 10 outgoing looks through one of the FPGAs. So we proceeded to the next level by similarly offering an all to all interconnection among four QFDBs, thus forming a tightly interconnected quad. Such a quad of QFDBs, in an all to all topology, is demonstrated in Figure 4.6, where QFDBs are depicted as circles at the vertices. This figure manifests that each QFDB gets interconnected through a single "Network" FPGA, which engages 6 out of the 10 available links for implementing this quad, again by deploying pairs of links per connection.

Based on an analysis of the possible applications that could be executed on top of our system, and mainly in order to make the best use of all transceivers currently available by the QFDB architecture, we concluded with the most favourable

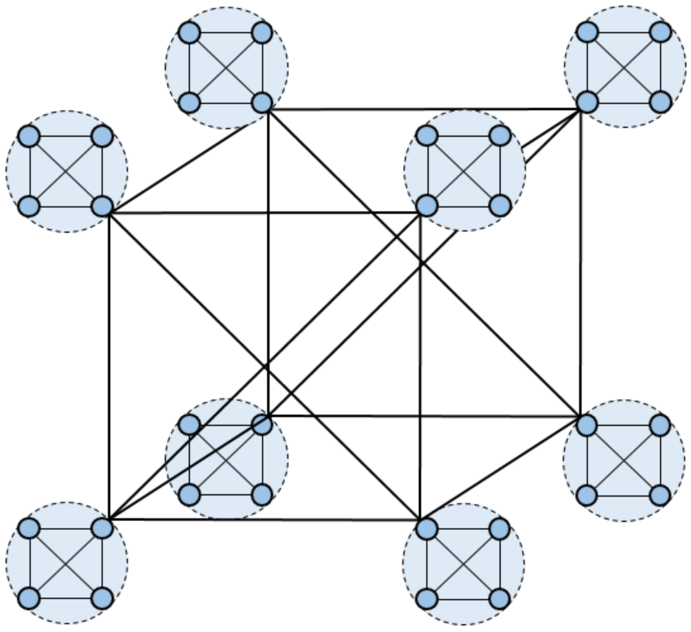


FIGURE 4.8: The 3D Cube with enriched connectivity

scheme at the current level being this all-to-all quad-QFDB interconnection. As each FPGA device, of the flavor used for the QFDB assembly, incorporates 16 transceivers in total, and as 6 of those where already deployed for the intra-QFDB connectivity, 4 still remain available to be used.

Moving on to the next level, and based on the still available links of a QFDB, we proceeded by interconnecting two such quads, leading to an interconnection scheme as that presented Figure 4.7. This actually comprises an unfolded cube. The same topology is presented in a three-dimensional way in Figure 4.8, where we more clearly visualize the formed cube topology. Actually, it more specifically forms a cube that also included with some extra diagonal connectivity. We can refer to this as an enriched cube topology. If one ignores the diagonal connections, something that can also be done at the application level, a user can actually operate on a more symmetrical and well known cube topology.

However, by adding those diagonals, the maximum intermediate hop count for any two nodes falls from 2 down to 1. We should note that this interconnection is completely analogous to the alternative that deploys the four internal diagonals of the cubes instead of the four we deployed at the two opposite faces. Furthermore, since in the depicted topologies not all 10 external QFDB connections are utilized, we could also create topologies of tighter interconnected cubes, even alternatives close to or even a complete fully interconnected cube, and thus more QFDB pairs

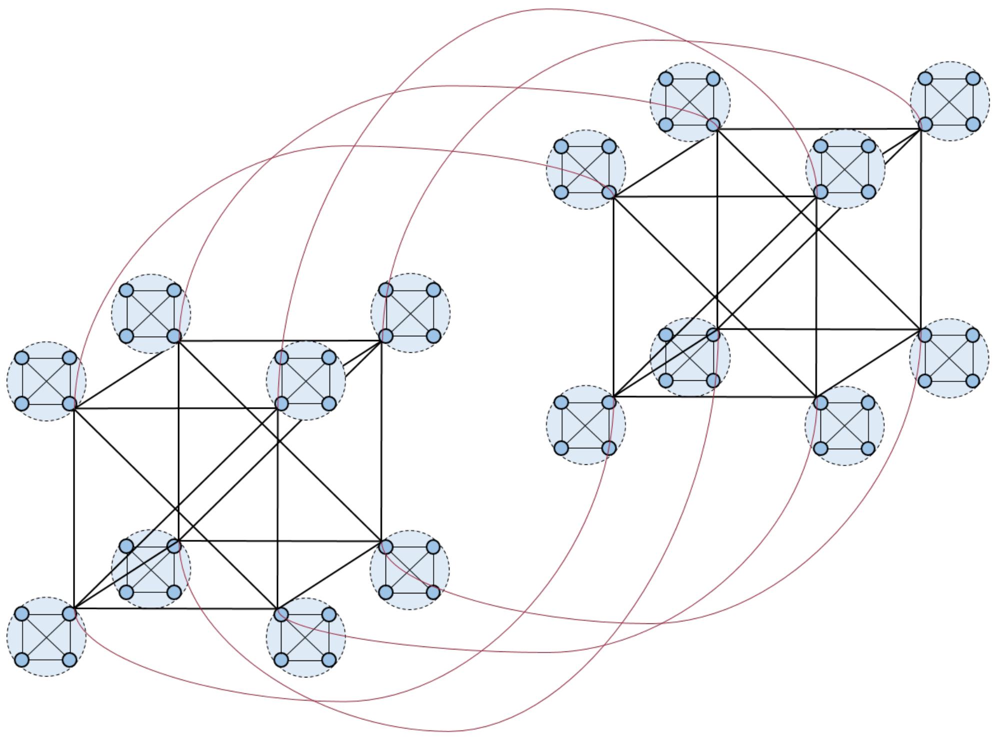


FIGURE 4.9: A 4D Hypercube with enriched connectivity

with distance equal to 1, i.e. with zero intermediate hops.

It is meaningful however to record at this point a simple metric on hop count versus link costs. We were able to reduce maximum intermediate hop count from 2 down to 1, by just adding these $2^2 = 4$ diagonals. However, if we wanted to go on to the last step, i.e. reducing maximum intermediate hop count from 1 down to 0, we would need $(2^2)^2 = 16$ diagonals. Based on the moderate count of QFDB link availability, deploying 4 diagonals seems as the most sensible and effective solution.

As we want to move on to the next level, we want many such cubes to get interconnected, in order to form a topology that can grow as desired. So, it is more important to use the remaining QFDB-outgoing links to allow further interconnects. So we proceed to the next step by creating a hypercube. This is accomplished by suitably interconnecting a pair of cubes, and is depicted in Figure 4.9. Each of the cubes encompasses the –optionally enriched– cube topology. This leads to a hypercube with a maximum intermediate hop count of 2, instead of 3, as would be the case without the enriching diagonals. A more formal analysis on this hypercube variation can be found in Appendix F.

Furthermore, a scaled-up platform can also be implemented, by using all the

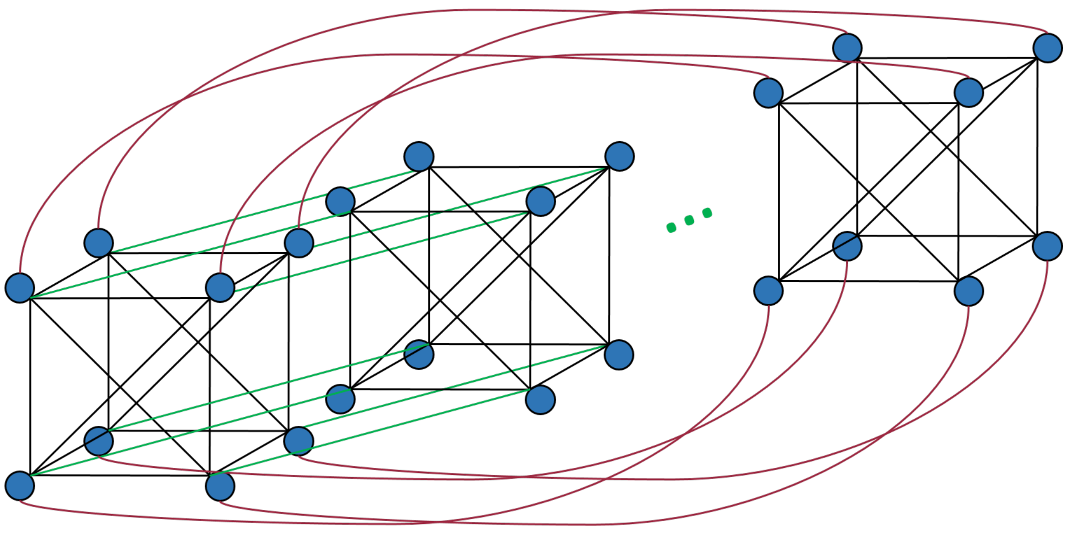


FIGURE 4.10: Interconnecting many Cubes

links offered by each QFDB. For example we can use the remaining available links to combine multiple cubes, as for example by creating a topology of a ring of interconnected cubes. Such a ring of cubes is presented in Figure 4.10. In fact, this can be considered as a 3D-torus variant, with two of the dimensions remaining narrow. Furthermore, other more standardized forms of 3D-tori can also be supported, as it has been demonstrated in [62, 94].

4.1.3 The earlier stages of the prototype, deploying QFDBs on mini-Feeder boards

A QFDB board comprises a system on module that plugs through a single connector, which provides power as well as all the desired interfacing. For our initial developments, we mounted QFDBs on top of unit carriers called Mini-feeders. These carriers essentially provide power to the QFDB, and formulates connectivity to the board. They feature 10 SFP+ cages, which allow us to interconnect the external high-speed serial links of different QFDBs together, at speeds up to 10 Gbps. Finally, these Mini-feeder feature a 1GbE PHY that enable the management of the QFDB, together with two UARTs. In our latest experiments, we moved on to using our custom and much larger carrier board, that is presented in the following section.

Before the custom baseboard was available, an in order to effectively evaluate scaling, we had to move on with scaled-up implementations and execute respective

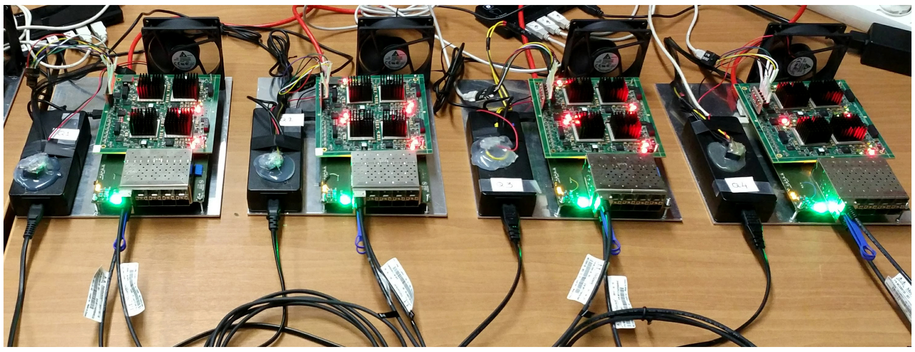


FIGURE 4.11: 4 QFDBs sitting on 4 mini-feeder boards

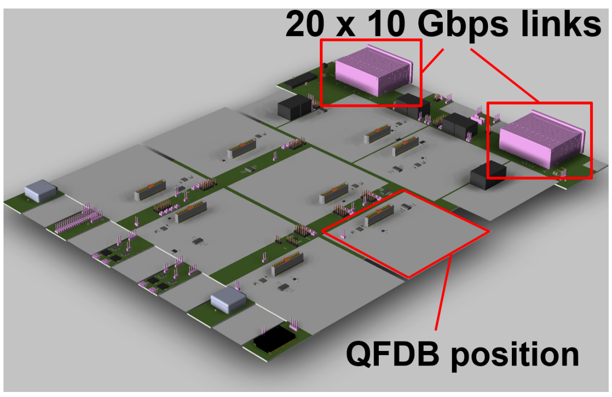
measurements, proving that the unification of FPGAs scales gracefully. We thus build a platform that incorporates 4 QFDBs (16 MPSoCs) as seen in Figure 4.11, each fitted on mini-Feeder boards, and interconnected through SFP+ cables.

Various interconnecting topologies have been taken into consideration. All the building blocks used, as well as the implemented architecture, are designed with the ability to give a high connectivity approach, so eventually, we utilized the available links to implement the full interconnection, i.e. an all to all connectivity, just as the one depicted in Figure 4.6. A lot of work was devoted in order to properly deploy, verify, configure and efficiently use the mini-feeder boards

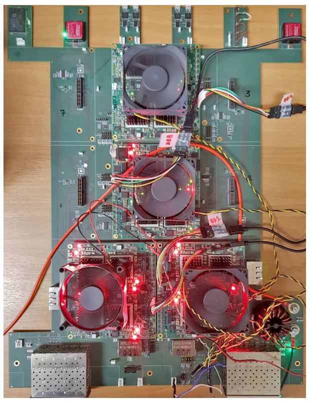
As already explained, intra-QFDB connectivity is realized through a pair of 16.25Gbps GTH transceivers for each connection, i.e. with an aggregate throughput of 32.5Gbps. One of the four FPGAs, designated as the "Network FPGA", offers "outside world" connectivity through ten links of 10 Gbps each, i.e. with an aggregate of 100 Gbps. These are driven through ten corresponding GTH transceivers in the network FPGA, adjusted to this speed due to the mini-feeder's SFP+ connector and the passive copper cabling limitations.

4.1.4 The Baseboard: 32-FPGAs in a 1U Chassis

In order build a prototype that exhibits the highly parallel UNILOGIC architecture, we designed a custom board referred to as "baseboard". This is a 52.8cm x 41cm dimensioned board, designed for and fitted in a standard 1U, 19-inch chassis of 420 mm width and 550 mm depth. As graphically presented in Figure 4.12a, it hosts 8 QFDBs, introduces them in a tightly interconnected manner, and provides them with all the needed interfacing, including UARTs, 1Gb management ethernet and as many as twenty external 10.3 Gbps SFP+ links.



(A) Annotated baseboard design



(B) A partially populated baseboard, hosting 4 QFDBs

FIGURE 4.12: The specially designed baseboard, densely hosting 8 QFDBs and offering tight interconnection

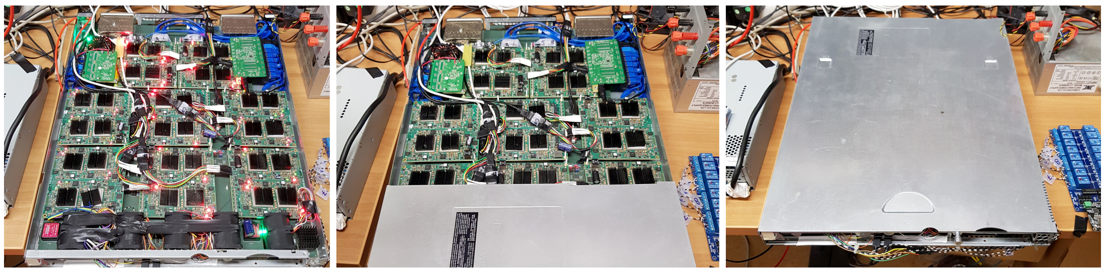


FIGURE 4.13: A fully populated baseboard, getting enclosed in a 1U chassis

A real life photo a baseboard laying on our lab bench can be seen in Figure 4.12b. It shows a partially populated baseboard hosting 4x QFDBs, allowing most of the underlying baseboard to be seen. A fully populated baseboard can be seen in Figure 4.13. This is hosting all 8 QFDBs and is placed in a standard 1U chassis enclosure. Fans in the front and back side of the enclosure provide the required cooling, so, as seen in the photo, the fully populated and functional blade can be completely covered.

While the QFDB boards are densely placed in a jigsaw puzzle fashion to provide the smallest achievable form factor, the inter-QFDB multi-gigabit interconnections are designed to support symmetrical, easy to deploy and flexible topologies, as the ones presented in the previous section [26]. In Figure 4.14, the QFDB slots are conceptually repositioned, in order to clearly present the offered connectivity; the two quads of QFDBs, left and right, are internally connected in an all to all manner. Then, these two quads are interconnected in a way that realizes the aforementioned

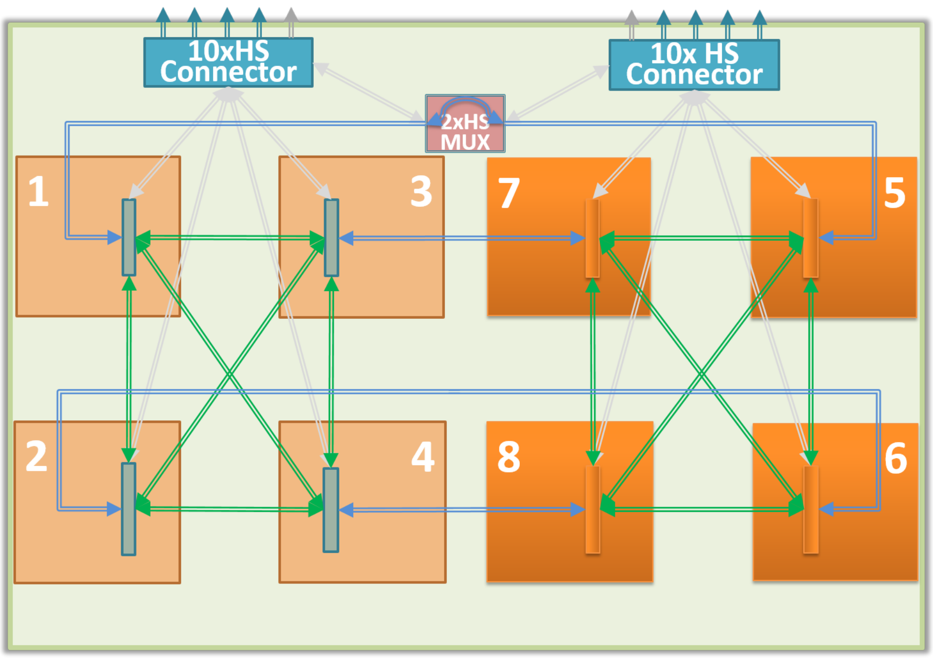


FIGURE 4.14: Baseboard's connectivity diagram

"enriched" cube topology of Figure 4.8. This way, any QFDB on the baseboard, ends up connected to any other either, directly or through a single intervening hop.

As for external connectivity, every QFDB is presented with at least a pair of 10.3 Gbps external links, aggregating to a total of 20 SFP+ connectors. Properly interconnecting through these connectors, varying topologies can be formed, as the ones in Figures 4.9 and 4.10. Six out of eight QFDBs are offered with two such links, while two of them can be selectively offered with an extra pair of links to the outside world. An intervening high-speed multiplexer at that point, as seen at the top of Figure 4.14, can be dynamically configured to dispense extra outgoing links to these QFDB slots, one to each quad. This flexibility leads to a wider range of topologies that can be supported. As these SFP+ links can be selectively deployed to enhance either inter- or intra-baseboard connectivity, it also addresses the need for various torus alternatives, as well as lead to further enriched cube interconnections, already discussed in section 4.1.2. The interconnection paths of Figure 4.14 are again represented with double lines, as each denotes a pair of deployed transceivers. Each baseboard, when fully populated hosts, eight QFDBs, thus a total of 32 FPGAs and $32 \times 16GBytes = 0.5TBytes$ of aggregate DDR4 memory.

Finally the baseboard offers a wealth of configurable signals, which drive various pins of the QFDB and the hosted MPSoCs consecutively. These signals include

- the boot mode of the devices (jtag, QSPI or SD-card)
- the power-up switch
- Processing System reset
- selective Programming Logic (PL, i.e. FPGA fabric) resets
- a number of signals that connects to pins of the configurable logic of the FPGAs and can be used e.g. to give a designated ID

These signals are driven through I2C bus expanders, header connectors or both, while they can also be controlled through buttons and/or jumpers. A Baseboard Management Controller (BMC) can be used to control both the headers and the I2C bus. Popular alternatives include the microZed [6] and the Raspberry Pi [31]. Simple software running on such a BMC module can easily control e.g. the ID bits given to the FPGAs, the high-speed multiplexer, the boot mode, etc.

4.1.5 The Final Prototype

Adding more QFDB nodes, in order to better evaluate and exploit our UNILOGIC architecture properties, we reached to our final prototype, depicted in Figure 4.16. It deploys two fully populated baseboards, as the ones described above. So, each baseboard hosts 8 QFDBs, translating to 32 FPGAs (MPSoCs). Both baseboards are displayed "open", i.e. outside their chassis or with open lid, in Figure 4.15, so we can grasp a better look of the hosted QFDBs. Then, in Figure 4.16, the baseboard on the left is depicted with the 1U-chassis case lid closed, as was explained above through Figure 4.13, while the one on the right remains outside its chassis for better insight to hardware.

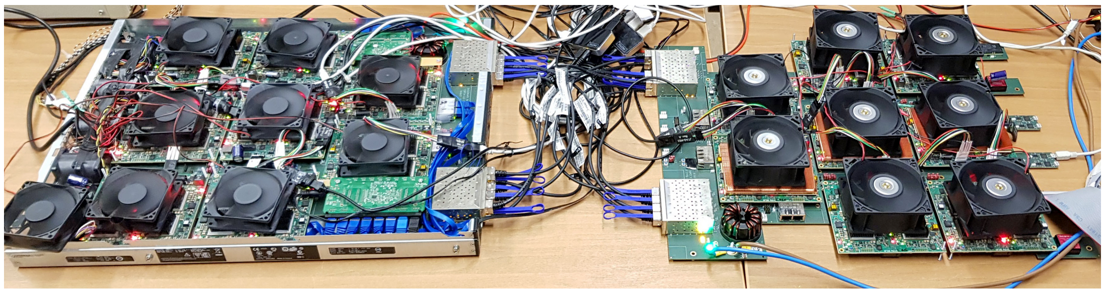


FIGURE 4.15: Two interconnected baseboards, aggregating 16 QFDBs

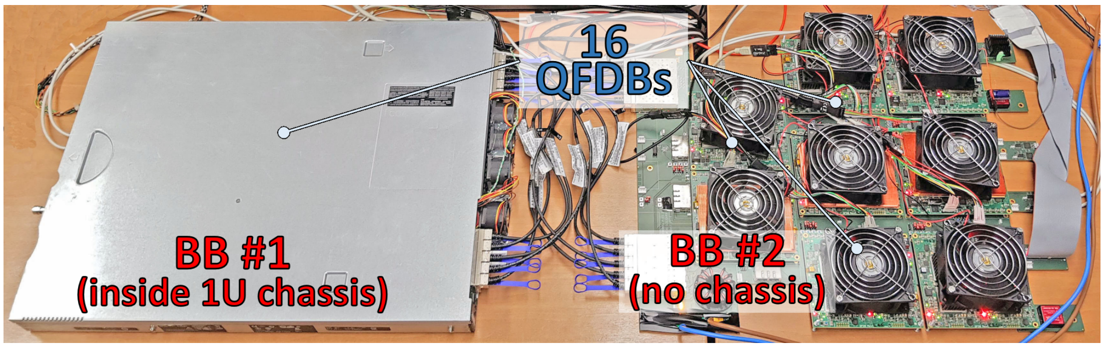


FIGURE 4.16: Two interconnected baseboards, with the leftmost inside a 1U-chassis enclosure

This prototype realizes the hypercube topology of Figure 4.9, while it hosts $2 \times 8 = 16$ QFDBs, $16 \times 4 = 64$ FPGAs, and a total $64 \times 16GB = 1TByte$ of DDR4 memory, operated at 2133 DDR speed (1066.66 MHz).

4.1.5.1 Infrastructure for Remote Prototype Management

Remote manageability is offered for all QFDBs hosted, including UARTs, Ethernet connectivity, jtag access, fine grained power consumption monitoring, selective power on/off, resets and more. For that reason, a dedicated Linux host machine (PC) was configures, which offers selective access to the board hardware and software. Remote access capability, originally allowed users from our lab at the Telecommunication Systems Institute (TSI) in Chania to effectively use the prototype. Eventually remote access is also granted to users from collaborating partners of the funding ECOSCALE project, such as Synelixis from Greece, University of Manchester from United Kingdom, Politecnico di Torino from Italy and Queen's University in Belfast. To offer remote low level hardware management, network operated relays allow users to selectively power on/off each QFDB separately, any selection of QFDBs at once, each of the baseboards, or the whole platform. Custom scripts offer simple commands for such operations, while corresponding scripts also offer a straightforward interface in order to separate among jtag controllers per QFDB, program QSPI memories, monitor FPGA through chipscope, etc. Also, depending on the targeted configuration, all 64 FPGAs of the prototype have their attached QSPIs properly programmed. This way on power up all FPGAs get configured with valid bitstreams, and the processing systems in turn automatically execute custom baremetal software in order to configure basic hardware settings such as chip ID and node ID. In brief, some of the most prominent operations, that an authorized remote user has access to are:

- turn on/off and power cycle QFDBs
- turn on/off and power cycle baseboards
- turn on/off and power cycle groups of QFDBs and/or baseboards
- have selective access jtag controllers of QFDBs
- reprogram QSPI memories per FPGA
- connect to and monitor UARTs
- differentiate among the management Ethernet connections
- run baremetal verification and micro-evaluation software on each MPSoC
- monitor hardware execution per FPGA, through the chipscope tool flow
- configure FPGA clocks through MPSoC management
- run high level software, e.g. deploying accelerators
- monitor power consumption per FPGA

4.2 Worker Architecture: FPGA Design & Addressing Scheme

In this section we demonstrate in detail the hardware design developed to implement the UNILOGIC architecture and allow any node in the platform to : a) access any accelerator core, b) (re)configure any accelerator slot and c) access any memory region, throughout the parallel platform. This allows for easy deployment and invocation of any available accelerator, according to demand, thus, greatly easing the FPGA platform's programmability. Mapping our high-level description of the UNILOGIC architecture, comprising of Workers (FPGAs) and Compute nodes (QFDBs), we demonstrate how each FPGA can be configured to become a self-contained Worker, while a full QFDB board can be tailored to comprise a highly effective Compute Node.

As part of this thesis, I almost fully devised the Worker architecture and moved on with its implementation on the Xilinx MPSoC. Devising the block diagram was both the first thing to do, as well as an ongoing process. Concerning the scheduler, I was involved in the specifications, while the HDL implementation was carried out by Chalmers University. I was then highly involved in the deployment, debug and upgrade process, as the scheduler was integrated in the FPGA implementation which was fully devised by me thought the whole UNILOGIC realization. I also devised the investigation on the Accelerator Slot Count per Controller. The PGAS addressing was also solely devised by me, as well as the addressing scheme under the MPSoC restrictions. The whole AXI translation scheme was also devised by me, as well as the HW translation tables and the SW for their pretranslated addressing setup. The memory port virtualization was also fully devised as part of this thesis, as was the selectively hierarchical AXI infrastructure and the related exploration. The conv-ID module was adopted from the EuroServer project as it was also part of the UNIMEM architecture by FORTH, while is was altered, fine-tuned and debugged for the more demanding purposes of the UNILOGIC infrastructure. Likewise, the C2C module was adopted from the UNIMEM implementation by FORTH, however it was upgraded to the newly deployed GTH transceivers, was debugged and upgraded for higher speeds now reaching 16.3 Gbps, while specific clock master/slave versions were devised to facilitate extensive activation of transceivers in each FPGA.

4.2.1 FPGA (Worker) Block Diagram

In this section we present an abstract view of the Worker (our Zynq UltraScale+ MPSoC device), focusing on the reconfigurable logic, i.e. the FPGA part of the device, that implements the UNILOGIC architecture. To keep the view more comprehensive, at this point we omit depicting blocks that would otherwise need deeper elaboration on our investigation concerning the architecture. Interesting parts of this analysis will be presented later in corresponding subsections. Going through the basic components, we should have in mind the core aim of the architecture, i.e. system wide unification and virtualization of hardware resources, as well as seamless access to any memory location, while realizing this globalization in a highly scalable manner.

Figure 4.17 provides the abstract view of such a Worker. We will need to explain how the depicted components interoperate so as to offer the required functionality. The shaded region denotes internal MPSoC device components, while for simplicity only the DDR memory is depicted externally. At the upper left of the figure, designated as the *ZYNQ MPSoC* and colored grey, is the Multi-Processor System of the device. It includes four ARM A53 cores, a DDR controller, DMA engines, while it interfaces to the rest of the chip, the FPGA part, through a number of master and slave ports, adhering to the AXI protocol. The off-chip *DDR memory* has a 64-bit bus, that has been configured to operate at 2133 DDR rate (1066.6 MHz).

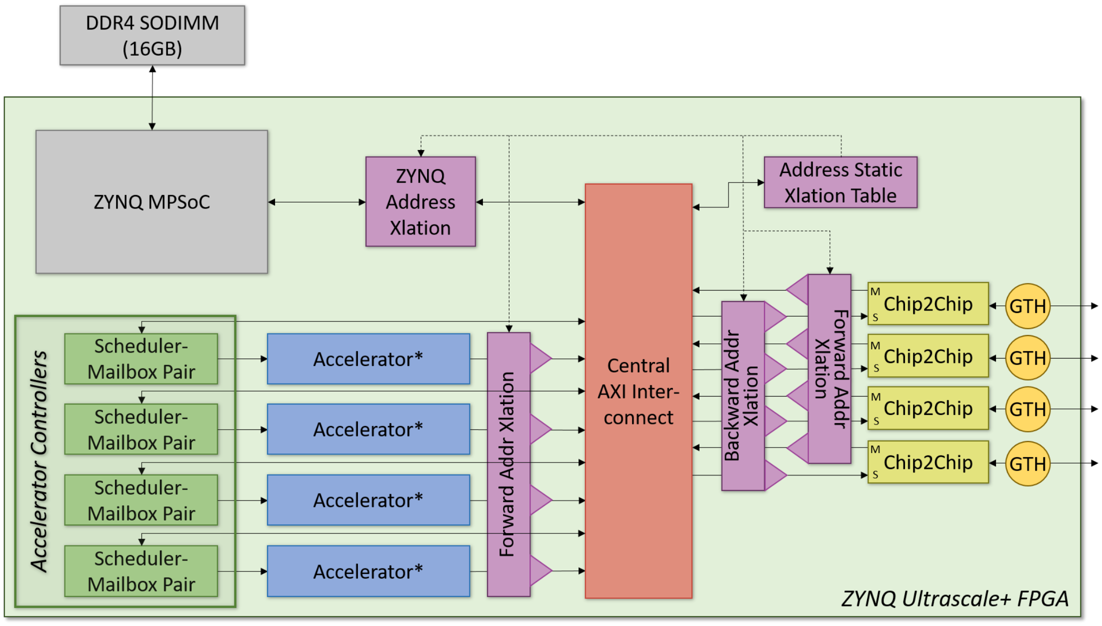


FIGURE 4.17: An abstract block diagram representation of a node's main configurable architecture

Looking in the FPGA's reconfigurable fabric, in blue color we see the *accelerators*. Accelerators can be built through the standard vendor tool flow, and need only adhere to the Scheduler interface. Each such blue block can actually correspond to an accelerator slot that can be configured with any different accelerator core, even at runtime, under dynamic partial reconfiguration. Furthermore, these slots can be dynamically combined, depending on accelerator size, so that two, three, or even all four can be dynamically merged to host a single accelerator core.

Before each such accelerator slots, on the left, stands a dedicated *Accelerator Controller*. This actually comprises of a pair of blocks. A *virtualization scheduler* is deployed to manage accelerators and organize task execution. The scheduler is coupled with a *mailbox*, which can accept and store commands, so that they can be processed at an appropriate point in time. Each such accelerator controller pair is depicted in green. These paired components will be later explained in more detail. Access to any of the accelerators, has to go through its coupled controller, which is done by just accessing the proper addresses in the global address space. The schedulers' attribute of software configurability, becomes of high importance, especially as the blue accelerator slots can be replaced by partial reconfiguration slots, allowing deferred accelerator types to be hosted. And also with the ability to perform the swapping at runtime. Also of note, when partially reconfigured, the apposite controller need to take charge of the combined slots, allowing bigger accelerators to be properly managed, no matter if they occupy a slot pair, or even merge three or all four slots, depending on the resources required.

On the right side of the figure, depicted in yellow, are the *chip-to-chip (C2C)* IP blocks which are associated with the multi-gigabit GTH transceivers. The C2C is responsible for adapting the intra-FPGA AXI interconnection protocol into the protocol used by the transceivers, and offers off-chip communication. The transceivers are FPGA vendor specific hardware primitives that can achieve highly efficient communication through serial links. In our case, they reach up to 16.3 Gbps, while achieving low latency. The C2C logic block presents them to the rest of the FPGA components, as an address mapped peripheral. So a processor, as well as any FPGA component, e.g. the hardware accelerators, can initiate read/write transactions that address C2C modules. The components of each Worker can thus communicate seamlessly, through the serial GTH transceivers, no matter if logic resides on the same or different FPGA. Hence, the system's resources appear as if they constituted a single, vast Worker. In the example of Figure 4.17, four such blocks are depicted. This can correspond to a miniature topology that designates three of the corresponding connections to each of the QFDB's three remaining FPGAs, and a fourth one to a remote board. More details on the C2C are presented later on.

In the middle of the FPGA logic, a *central AXI interconnect* infrastructure, depicted in red, is deployed and configured in order to allow all components to communicate efficiently. We have adopted the Xilinx AXI protocol in our architecture, to harmonize both the local and remote accesses, correspondingly for intra- as well as inter-FPGA communication. AXI allows for a standard, uniform, and efficient way to interconnect hardware modules. It can support bursts, variable address-bus and data-bus widths, multiple outstanding transactions and user defined signals, while its efficiency as an on-chip interconnection framework has been widely demonstrated [102, 73, 103, 74]. On the other hand, AXI is not designed for communicating across chips, and for providing the desired interoperation. This restriction can become even more puzzling, when more complicated intercommunication topologies have to be supported, as in our case. This poses certain obstacles, while addressing and overcoming these obstacles is part of the UNILOGIC architecture, with more details presented in the AXI related sections below.

Complementing the Worker architecture, scattered in the block diagram and shaded in purple, stand the *addressing related blocks*. These blocks are assigned with the task to translate the addresses produced either by the processors or by the accelerators. They also intervene in any transaction leaving towards or arriving from the boundaries of the FPGA. Addresses generated by the processor aim to either invoke a local or a remote accelerator via accessing its designated controller, or to directly access data that reside in memory, anywhere across the unified system. The accelerators, in turn, generate addresses to request data from memory across the system, and likewise to send back the results. The translation blocks have to properly translate all of these transactions. At the same time, as denoted by the dashed lines, they are being orchestrated by a hardware translation table that is software configurable through software running on the processors. The necessity of such intervention in the transactions flowing through the system, and the enforcement of proper address translation, is of primary importance for our architecture implementation, involved significant effort and investigation, and will be presented in more detail below.

4.2.1.1 Details on the Block Diagram

Before moving on with elaborating on the most important parts of the block diagram, comprising the basic infrastructure of a UNILOGIC Worker, we believe that a more detailed perspective of the diagram can prove beneficial, offering a closer to the UNILOGIC designer point of view. A sample of such a block level view on the implemented architecture is shown through a Xilinx's Vivado tool block diagram snapshot, depicted and annotated in Figure 4.18. At the bottom of the Figure, legend bars provide insight on which blocks accommodate UNILOGIC functionality, UNIMEM, or both of them in tandem. On the top left we can see the Accelerator Controllers, i.e. the pairs of mailboxes and schedulers. These are attached to accelerators a bit to the right, which then get serviced through

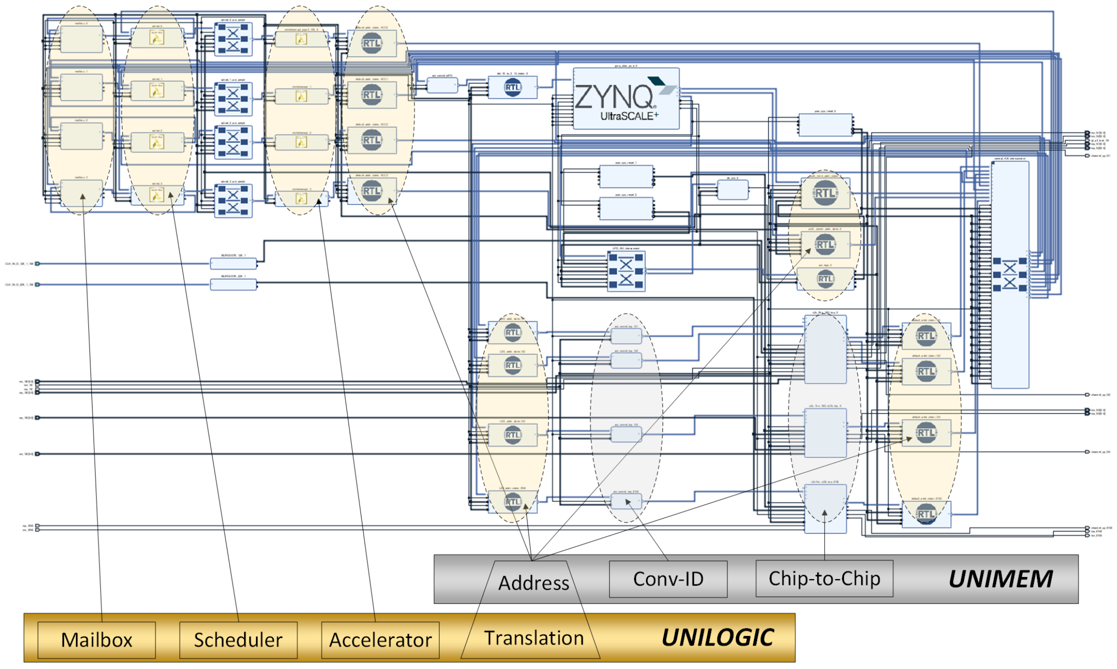


FIGURE 4.18: A block diagram perspective of a single Worker (FPGA) as seen in Xilinx's Vivado tool, highlighting main components of the collaborating UNIMEM and UNILOGIC architecture

blocks that provide the UNILOGIC address translation mechanism. The accelerators blocks are substituted by partial reconfiguration slots, not disclosed in this view for simplicity. At the bottom of the diagram we see the the C2C blocks that offer external connectivity, standing in between address translation blocks, and provide the forward/backward address translation we have seen purple-shaded in Figure 4.17. The conv-ID portion augments the UNILOGIC address mechanism, mainly for cross-chip communication, and will be presented later. Above these, blocks offering UNILOGIC translation for the processor initiated transaction are also depicted.

As this view of the block diagram can give a better insight on the architecture implementation, if not to the architecture itself as well, we aimed to further disclose more details on the designs employed in our final prototype. These include all the UNILOGIC blocks, along with blocks related to dynamic partial reconfiguration. For that reason, and as the discussion is of a more technical nature, this information is presented in Appendix B

4.2.2 Accelerator Controller: Virtualization of the Accelerator Slots

Starting our more elaborate description from the bottom left of our abstract block diagram (Figure 4.17), we visit the Scheduler-Mailbox pairs. This block comes to support one of our principal objectives, when building an architecture that efficiently supports distributed and shared reconfigurable resources, i.e. ease of programmability. This involves automating the distribution and dividing of big tasks into smaller ones, which can be executed by the reconfigurable hardware accelerators, while being transparent at the application level. Such a goal pertains to a virtualization infrastructure which, at Worker-level, should target to virtualize the acceleration resources of a single FPGA, and then at System-level, upscale by similarly operating on a multi-FPGA unified environment.

The virtualization block is realized through the *"Mailbox"* - *"Scheduler"* pair. The hardware Scheduler [79] can be connected to, and thus control, a varying number of hardware accelerators. These accelerators, typically built by the standard tool flow, have an AXI-lite slave interface to accept configuration arguments and start/stop commands. The Scheduler gets hooked to this interface. Accelerators, also incorporate an AXI master interface that they use to access memory. The reasoning behind having the Scheduler gain control through this interface, is that it is supported by the standard High-Level Synthesis (HLS) tools of the FPGA provider (i.e. Xilinx). Now the accelerator is "hidden" behind the Scheduler, along with any configuration-specific details. The Scheduler in turn can be configured, through the Mailbox, to operate on any kind of available accelerator.

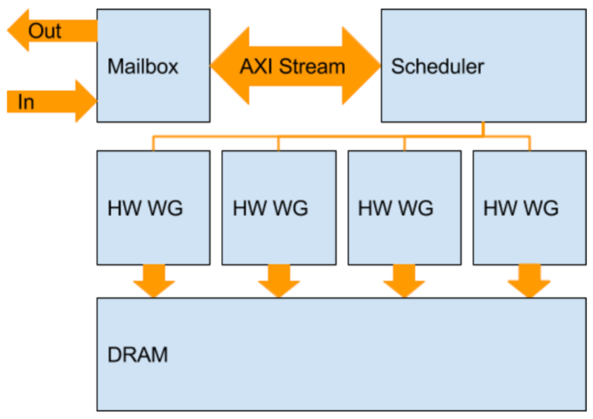


FIGURE 4.19: Overview of the virtualization block and its interfaces

In more detail, to configure and trigger an accelerator, a set of commands has to be sent. These commands are initially generated by the application running in the processor and sent over to the Mailbox module that resides in the reconfigurable logic. By introducing the mailbox hardware primitive, we can setup, start and stop hardware accelerators that reside on local or remote workers. Then, our novel Scheduler hardware module, dequeues the configuration commands from the mailbox, and once a complete set of parameters is received, it can invoke the corresponding hardware accelerator. This is a key feature, as it makes the Scheduler "generic"; generic in that a single implementation of the Scheduler will work with any number of different hardware accelerators as long as a suitable set of configuration commands is driven through the mailbox.

The Scheduler is additionally capable of dividing large accelerator tasks to smaller ones, that exactly fit into the targeted reconfigurable hardware resources. Thus it can separate and issue a series of smaller subtasks which, when combined, execute the larger task originally requested. Furthermore, as a single Scheduler can be connected to multiple hardware accelerators at the same time, it is able to orchestrate all available resources so as to execute a volume of subtasks, even originating from different applications that ask for the same hardware resources.

The Scheduler works in a logic loop as follows :

- If no work is ongoing, check the mailbox for a new command.
- If work is ongoing, look for the next free accelerator to place the next computation subtask.
- If the work assigned to an accelerator has been completed, put a message on the outgoing mailbox to indicate that this accelerator has completed computation.

This functionality is graphically presented in Figure 4.19, where accelerators are designated as Hardware Work Groups (WG). The Mailbox on top incorporates two FIFOs for write and read operations. Acceleration request commands are first issued to the mailbox (to be then propagated to the Scheduler) by writing to the incoming FIFO. Any required status information can be read from the mailbox's outgoing FIFO, which actually receives data from the Scheduler. The Mailbox and the Scheduler intercommunicate through a dedicated AXI Stream interface.

Each Scheduler command is a multi-word packet, with the number of words per command varying according to the specific accelerator type. The main commands are:

- "Configure Type": Containing information about how the work of a new type of accelerator should be scheduled, how the available bitstreams are addressed etc.
- "Configure Setup": sets the actual number of accelerators and their address mapping.
- "Accelerate": describes the aggregate acceleration task through proper arguments.
- "Kill Scheduler": terminates Scheduler execution.

The Scheduler in turn responds with messages through the mailbox. These messages notify the initiator application for configuration completion, work completion, as well as various errors that may occur, such as wrong opcodes etc.

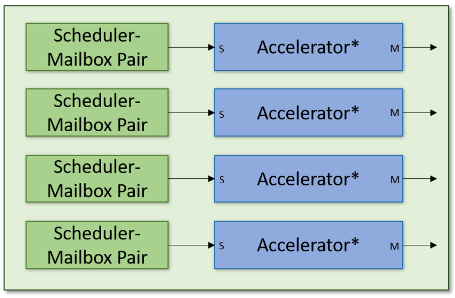
So, in a high level summary, the complete virtualization block works as following:

- Software running on a PS instructs the Scheduler to start, by just writing a word at a specific address.
- Then, by writing to the mailbox address, software indirectly configures and starts the accelerators.
- The Scheduler transparently slits and orchestrates tasks, based on available accelerators
- In parallel, there is always status (debug) information available in the Scheduler status registers.

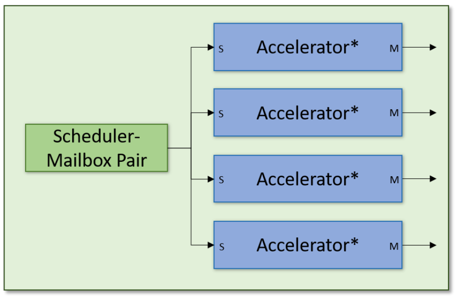
As a result, by just using the memory mapped Mailboxes and Schedulers, an application can use simple load/store commands that will end up to transparently configure/start/stop/check any accelerator in the system. As described in the next section, this virtualization scheme is still effective, no matter where the reconfigurable resources reside, either locally or remotely (i.e. to a distant node) by properly integrating our virtualization approach within the UNILOGIC communication infrastructure.

4.2.2.1 Investigation on Alternative Setups for Accelerator Slot Count per Controller

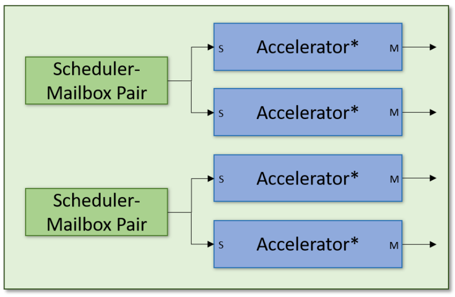
Before getting to more details on the addressing scheme, we would like to discuss some accelerator controller related alternatives that we have investigated on and are available to the designer that implements the UNILOGIC architecture. The



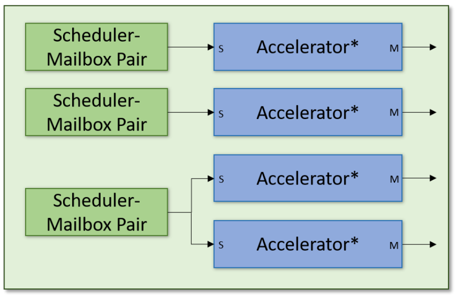
(A) One Scheduler per Accelerator



(B) A single scheduler for all Accelerators



(C) One Scheduler per Accelerator pair



(D) A mixed Scheduler approach

FIGURE 4.20: Various design alternatives concerning the varying number of accelerators (or partial reconfiguration slots) that can be controlled by a Scheduler

Scheduler as discussed can accept instructions through the mailbox, corresponding to the current accelerator it has to control. These instructions can correspond to a very large acceleration task, consisting of smaller subtasks, and the Scheduler has the ability to send the smaller tasks one after the other. It can furthermore dispatch these subtasks either to a single or to multiple hardware accelerators. Hence, controlling in parallel many accelerator slots, it can assign each following subtask to the next available accelerator, i.e. to any of the attached ones that has just reported to have finished its previously assigned subtask. They are all operated in a parallel fashion, orchestrated in hardware by the Scheduler, and have to be of the same type.

In our final and presented UNILOGIC prototype, as already seen in Figure 4.17, we have chosen to couple a single scheduler/mailbox pair with a single accelerator. This case is also depicted in Figure 4.20a. However, in Figure 4.20 we also see other possible alternatives. Figure 4.20b, depicts the other end, which is a single Scheduler for all four accelerators. In this case, all accelerators of the same kind can be controlled and run in parallel by the Scheduler, with no software intervention.

The Scheduler itself can support a much higher number of accelerators, however the examples depicted are based on the four accelerators per FPGA setup, as supported by our implemented prototype, for ease of understanding. This one-to-many solution has been already implemented and tested. We have successfully used it on our first, proof-of-concept, single FPGA prototype [75]. Some analysis on this will be presented in the evaluation section 6, however it is important to note here that we have even analyzed hardware architecture realizations with up to 12 accelerators controlled by a single scheduler, with encouraging results. It is also important to explain, that even under this one-to-many configuration, it is not prohibited to attach diverse accelerator cores. However, the software running should be cautious on properly configuring the scheduler, as it should only be guided to control a single type of accelerators each time. For example, we could configure the first three slots with accelerator type "A", and the single remaining slot with accelerator type "B". The scheduler can then be configured for type "A" accelerator controlling, and accelerator count equal to 2. After the requested job is finished, the scheduler's configuration can be altered to control type "B" accelerators, and count equal to 1.

Finally, Figure 4.20c presents a solution in between the previous two, with one accelerator controller per couple of accelerator slots. This allows for more flexibility, with moderate hardware complexity due to the reduced hardware blocks used. Finally Figure 4.20d presents a mixed solution. It's up to the designer and based on the predicted accelerator utilization and execution patterns to select the best fitted solution.

In our final prototype we deployed discrete accelerator controllers per accelerator slot, offering the highest flexibility by allowing diverse accelerator cores to be deployed and run in parallel. We have carried out thorough investigation, and implemented the solution of Figure 4.20a, as this a) allows the highest level of diversity and parallelism, i.e. four different accelerators all running in parallel, as directed by the designated controller, b) can support the most flexible runtime partial reconfiguration, as any number of adjacent slots is allowed to be merged, and merging can have any starting slot, thus also providing the means for resource defragmentation through accelerator migration, as will be presented later in section 4.3.

It is important to note that this choice, while being the most profitable, nonetheless entails the highest realization effort. This is so because a) hardware complexity increases, as more scheduler/mailbox pairs need to be encompassed and controlled through the PGAS, and furthermore this means that they have to be accessed through the central AXI interconnect, which requires special care in order not to excessively grow, threatening performance, b) software applications need to become a bit more fine-grained, in that when many accelerators of the same kind are configured, a big task needs to be software divided to more sub-tasks, and be sent over to 4 controllers per FPGA, instead of four times larger and less tasks, and c) although importantly flexible, such a setup does not easily favor performance metrics, and more design effort was devoted for performance to scale gracefully, as more accelerator controller invocation messages need to be sent over, and special care such as low latency and multiple supported transactions need to be efficiently deployed. So it should be clear that this choice was made in order to archive the best result, although the realization effort was greater.

Finally we would like to report that the final, 4 accelerator slot setup was chosen as our proposed middle ground solution, based on accelerator sizes, and deploying accelerator core alternatives of varying optimization levels and corresponding sizes. Also as the static logic overhead is kept modest, most FPGA resources are left to the accelerator slots, therefore allowing for considerable logic per slot. This way, each of four slots can fit satisfying accelerator sizes, thus not causing logic fragmentation. Furthermore it becomes an effective setup under dynamic partial reconfiguration, and in the case when slot merging has to occur, configuring large accelerators that do not fit in a single slot. Lastly, four slots can get optimally coupled to the four parallel ports of the PS, thus driving a discrete fast path to DDR memory for any of them.

4.2.3 PGAS: The realized Addressing Scheme

Before visiting any other components of the architecture's abstract block design, it is beneficial to present the global addressing scheme realized. We will start with highlighting its basic parameters such as the number of nodes that should be supported to allow system scaling, as well as the per-node memory region. Then we will clarify the need for the aforementioned translation blocks. The addressing scheme of the UNILOGIC architecture, has to scale in an efficient way up to exascale related sizes, while each node has to include tens of Gigabytes of memory. The current implementation actually corresponds to a specific instance of the generic addressing scheme already demonstrated in Figure 3.4. In our prototype, each node is implemented so as to expose a 32GB address region, depicted in Figure 4.21. The exposed region includes the 16 GB of the DDR memory that is coupled with each MPSoC, and is supported by the current QFDB implementation. In the other 16 GB, a few MBytes are needed to address the Worker's peripherals, i.e. the logic blocks residing in the FPGA. In order to retain alignment per Worker, the remaining region is reserved, resulting to the more convenient 32GB-sized region. This way, MS bits of an address can easily designate a node, i.e. FPGA, and LS bits can designate the memory or peripheral.

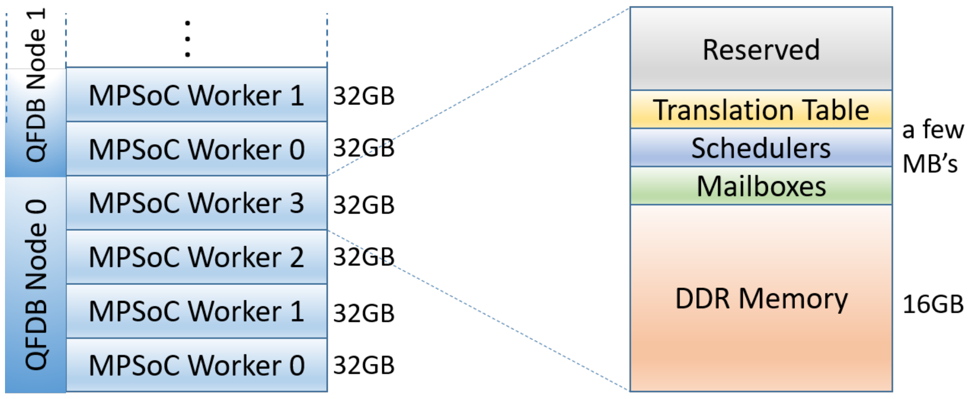


FIGURE 4.21: A portion of the UNILOGIC global address space, focusing on a node's address window

Based on this designation of low and high order bits, the resulting address for our prototype system can be seen in Table 4.1. The 32 GB region designated per Worker, needs 35 bits to be addressed, either to target its memory, i.e. 35th bit is 0, or its peripherals. The 6 MS bits specify a unique Worker (MPSoC): they include 2 Worker-ID bits (WID) referencing one of the 4 MPSoCs of a QFDB, and 4 Node-ID bits (NID) designating one of the 16 QFDB boards available in our prototype. 41 bits are needed in aggregate, while we can add as many NID bits as needed in order to deploy an increasing number of Compute Nodes, i.e. QFDBs, in our implementation as desired. The way the addressing mechanism is integrated in the overall architecture implementation imposes no limitation whatsoever to the width of the supported addresses, so that any number of Compute Nodes can easily be supported by just defining the proper NID field width. Encompassing 16 QFDBs aggregates $16 \times 4 = 64$ FPGAs, and $64 \times 16\ Gbytes = 1$ TByte of DDR Memory, all addressed through the PGAS, as designated by the 41-bit addresses. It is important that we surpass the 40 bit address width threshold, as it relates to an MPSoC addressing restriction that will be explained below. By getting above this threshold, we had to implement and verify certain hardware mechanisms that allow the 40-bit capable Processing System (PS) transactions to access this 41-bit address region.

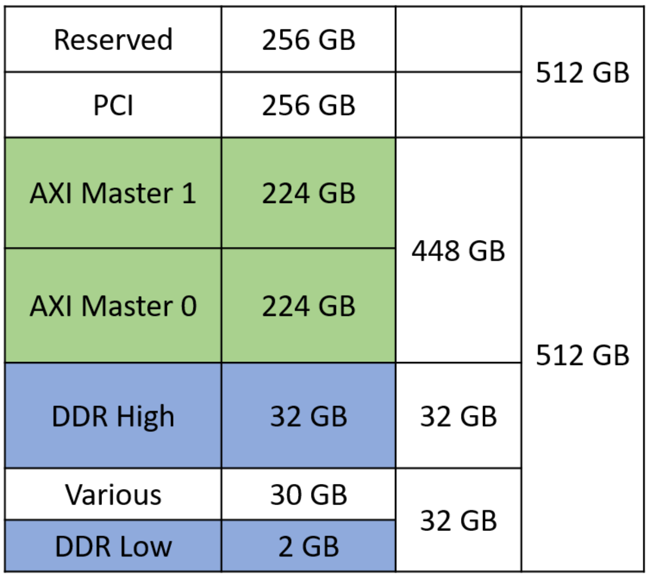
TABLE 4.1: The fields of the addresses used in the UNILOGIC System's global addressing scheme, for the 16-QFDB/64-FPGA platform

41-bit UNIMEM + UNILOGIC global address			
NID	WID	~mem/per	LS Address bits
4b	2b	1b	34b

4.2.4 Zynq MPSoC: Addressing Restrictions & Solutions

Looking again at Figure 4.17, at the upper left side stands what we designate as the ZYNQ MPSoC, which corresponds to the multi-processor block shortly described above. This block interfaces with the reconfigurable logic, i.e. the FPGA fabric, through multiple interfaces, all adhering to the AXI protocol. However, the actual maximum addressing space allowance for this fixed hardware block, is necessarily limited by construction. We will thus begin with presenting some entanglements that arise from using such an FPGA SoC, which therefore require proper handling. The addressing method we opt to implement in our prototype that gets comprised of such SoCs, will have to be based on the generic UNILOGIC architecture, ans likewise the PGAS scheme presented in section 3.2, with the actual PGAS addressing realized as will be presented below.

We will start our analysis by considering a new transaction initiated in the system. This usually starts at the Processing System (PS) of an MPSoC, and its destination is signified through a target address. he initiating processor would firstly need to communicate with the FPGA part of the MPSoC (PL) through the AXI-compliant interface ports. In this case the AXI addressing is confined



Reserved	256 GB		512 GB
PCI	256 GB		
AXI Master 1	224 GB	448 GB	512 GB
AXI Master 0	224 GB		
DDR High	32 GB	32 GB	
Various	30 GB	32 GB	
DDR Low	2 GB		

FIGURE 4.22: Address map simplified view

to a 40-bit address scheme, hence supporting up to a maximum of 1 TB physical address space. Fig. 4.22 gives a simplified view of how the MPSoC's address space is statically partitioned. In this statically configured address map, we have highlighted the regions of highest interest for UNILOGIC, while a more detailed analysis can be found in the respective Xilinx datasheets [126]). AXI, i.e. the Xilinx's interconnect protocol, comes along with the corresponding interconnection IPs [7] offered by Xilinx, i.e the configurable hardware modules that allow for efficient intra-FPGA communication among deployed logic blocks. Inside the PS, any transaction, either generated internally or originating from the FPGA's logic blocks, is routed based on this mapping. For example, if a transaction's address falls inside the lower 2 GBytes, it will be routed to the on-chip memory controller which handles it accordingly to access off-chip DDR memory. If an address falls in the AXI Master 0 or 1 regions, then it will leave the PS block through the corresponding AXI port and enter the Programmable Logic (PL), i.e. the reconfigurable FPGA fabric. Hence this address region is of main significance, as it pertains to the path that leads both to the local node's reconfigurable logic, i.e. local accelerators, as well as to remote nodes, i.e. through the C2C blocks to remote accelerators or memory of the system.

Stepping into more details, there exist some further restrictions imposed as to how this address mapping is implemented in the MPSoCs. Firstly, all the AXI Master regions of Figure 4.22 add to a total of 448 GB. This is the aggregate

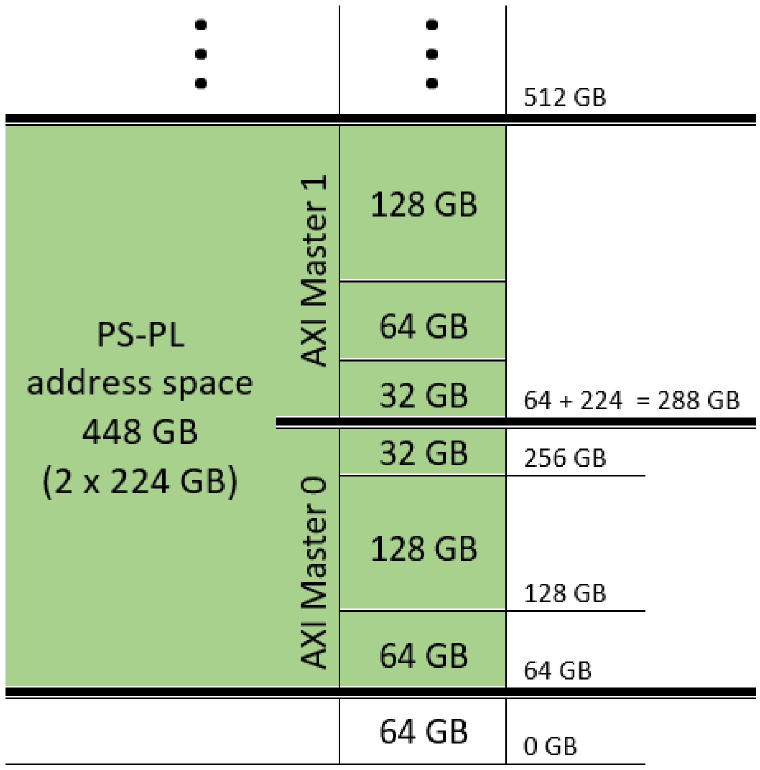


FIGURE 4.23: Alignment and Fragmentation of the PS to PL address space

memory region through which an MPSoC can access remote Workers and Nodes, i.e. by initiating transactions that will be passed on to the PL and then get forwarded through the multi-gigabit transceivers existing therein. We thus have to use it accordingly, in order to support a desirable and scalable number of nodes in the prototype. An related implication arises again due the MPSoC implementation and as the PS ports adhere to the Xilinx's AXI interconnection protocol: AXI imposes that every slave module targeted through a master port should be mapped in an address region that is power-of-two sized and correspondingly aligned. This is better demonstrated through Figure 4.23. When trying to map a peripheral e.g. to AXI Master 0, the largest consecutive address space available is just 128 GB wide. The next larger power-of-two size would be 256 GB which does not fit in this Master's space. Additionally, it is mandatory to have the base address configured to a multiple of 128 GB, i.e. 128G or 0x20_0000_0000 in this case., so that it becomes properly aligned. The same goes for a second similar range that can be utilized by AXI Master 1.

Added to the aforementioned limitation, a single AXI peripheral cannot be assigned a concatenation of two regions, e.g. a 64GB and a 128GB region. Thus finally an MPSoC's slave AXI peripheral can only be mapped to a maximum address region of 128GB. We should note here that using some tricks, a designer can bypass this "region-stitching" restriction, however not in an elegant way; one can, for example, use three intermediate PL peripherals that map to an AXI Master with a size of 32, 64 and 128GB respectively. Then, through an AXI interconnect, all these peripherals can map to the originally targeted PL block. This will end up to an indirect mapping for the targeted peripheral of the complete 224 GB region. This cumbersome design trick can extend the visible address space, but on the other hand it also increases both design complexity and critical path latency, with the later being of most importance for our architecture.

In order to device a solution we need to focus on a transaction, and thus a target address. This can be launched either by an accelerator, or by an application running on the PS. Accelerators on one side use addresses that are submitted by an accelerator controller, with both the accelerators and the controllers residing within the reconfigurable logic. Hence they can be built to directly use the global addressing scheme, i.e. 41-bits for our prototype implementation. A PS initiated transaction on the other hand, need to leaves the PS in order to target either a remote memory region, or any peripheral, and thus cannot adhere to the global address width, restricted due to its implementation as described through Figure 4.23. Since the PS has two main outgoing (master) ports, we can use one for memory accesses, and the other for peripherals. The limiting factor is the maximum of 128 GB continuous address region that can get addressed through each of those, as seen in Figure 4.23. Nevertheless, for our current implementation, we are able to allow full system addressing even by using just 64 GB per port. What is more, the solutions developed scales easily to much larger platform sizes.

For memory accesses, 64 GB suffice in order to access the entire memory of a single QFDB. On the other hand, since peripherals span just a few MBs of address space, 64 GB can suffice for accessing peripherals system-wide. As a result, we follow a twofold approach. To have a better understanding, we should once more look back at the address resolution of Table 4.1.

On one hand, for memory accesses, the whole 41-bit address of table 4.1 should be used to signify the target. Thus it cannot fit within the 40-bit PS addresses and even worse within the 36 bits of the aforementioned 64 GB region. We can, however, separate the 4-bit NID of an address, as seen in table 4.2, and store it

TABLE 4.2: The 'intermediate' memory addresses, as issued by the PS to the PL, aiding to target system-wide memory

Global memory address passed from PS to PL		
NID (QFDB)	WID (FPGA)	Peripheral
4b	2b	34b

into a special register in the reconfigurable logic, prior to issuing the corresponding memory access. The remaining 36 bits, which designate the memory address inside a QFDB, fit and can be issued directly by the PS. Then, both this address and the special NID register get properly concatenated to form a proper 41-bit global memory address. This technique comes both in a straightforward and a scalable manner. Separating just the NID easily fits to our general addressing scheme, while simply writing wider values to the NID register, allows for any number of nodes in the system, and, consequently, allows for scaling to any desired size. Furthermore, although this NID separation leads to a two stage addressing, it injects only minor latency. This is so, as minor delay is added with the actual transaction following in the minimum AXI allowable gap. Even more importantly this mechanism needs to be invoked rarely, because most often numerous memory transactions, e.g. through DMA accesses that actually require high throughput, are issued for consecutive memory accesses. And accesses towards a certain Worker, i.e. FPGA, and even more to a whole Compute Node, i.e. all four FPGAs of a QFDB, pertain to a single NID. So steering to a different NID happens once every many memory accesses. So this set-once/use-many two-step addressing scheme does not hinder performance.

TABLE 4.3: The 'intermediate' peripheral addresses, as issued by the PS to the PL, and targeting any peripheral system-wide

Global peripheral address passed from PS to PL		
NID (QFDB)	WID (FPGA)	Peripheral
4b	2b	30b

On the other hand, peripheral accesses are usually short and non-contiguous as they pertain to short configuration commands. However, peripherals themselves only span a small address region per FPGA. So, in this case, even a single 64 GB address window suffices to target all the peripherals system wide, including the NID/WID portion of the address at once. The 36-bits of the 64 GB window can be separated, as seen in table 4.3, to 6 MS bits for NID/WID designation (jointly named NWID), while the remaining 30 bits (1 GByte) more than suffice to target the designated node's peripheral. Proper address padding and concatenation is performed in a transparent manner, when such an access exits the PS and enters the PL. As for scaling, even deploying the current Xilinx MPSoC generation, we can: (a) map much less than 1 GB for peripherals per Worker as needed, and (b) utilize the complete 128 GB of the PS port AXI address window, and end up with many thousands of QFDB nodes that can still be directly accessed.

A last thing to note is the build-in fragmentation of the DDR memory address space that can be seen in the blue-colored regions of Figure 4.22. These regions allow any logic block residing in the FPGA to access DDR memory through the PS ports available. It is fragmented into two distinct regions, in order to facilitate prior 32-bit PS architecture backward compatibility. For such a compliance, a limited lower address space of 2 GB is mapped to memory, to fit with the confined addressing mode of the 32-bits versions. This is designated as "DDR Low". Memory addressing fragmentation could prove cumbersome when trying to have a uniform global address space in a multi-node system. We thus opted for a solution to offer a contiguous and aligned memory region, i.e. mapped to the lower address space. Unfortunately, the lower 2GB region cannot be omitted, in order to solely use the "DDR High" memory region for the mapping of our 16 GB total available memory. Hence, we devised a a custom hardware block that intervenes in the PS ports and translates all memory accesses on the fly, i.e. without adding any extra clock cycles to the memory access path. It transparently allows all the memory address space to be accessed by the "external world" as if it was contiguous.

4.2.5 Central AXI Interconnect: Overcoming cross-FPGA communication restraints

As we have already mentioned, the AXI interconnect offers many benefits for intra-FPGA communication. However, it is not a perfect fit for inter-chip communication. A notable entanglement emerges when deploying the AXI interconnect as a fundamental building block of the global interconnect. Consider a transaction initiated inside a Worker, i.e. MPSoC device, which tries to access some other remote Worker. In our case this has to route through the Central AXI Interconnect (the red block of Figure 4.17), using proper AXI address mapping. This mapping has to include an address range to map intra-Worker resources, and also three such identical ranges to map the other three interconnected Workers on the same QFDB. What is more, another range has to map to remote QFDBs. But what would the address be, to steer towards a QFDB's outgoing link? This situation is conceptually depicted in Figure 4.24. We have to somehow efficiently target the remaining hardware resources, but this addressing cannot be directly translated to a properly configured AXI interconnect. It depends on the topology, and on the specific QFDB we refer to in the whole system. This would have to include address fragments across all remaining address ranges, which makes it almost impossible to compose. What is more, it would additionally ruin any effort for hardware uniformity across Workers; any node would require a disparate and complex configuration.

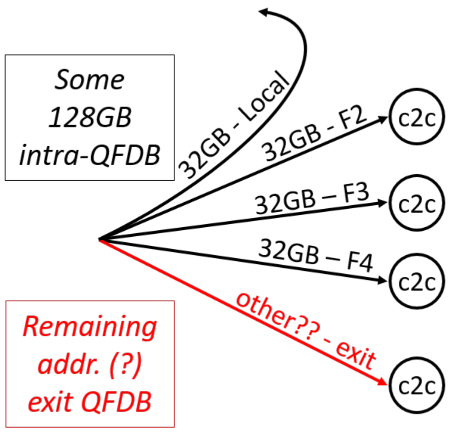


FIGURE 4.24: An AXI addressing empuzzlement

In order to overcome this problem we propose and implement a uniform hardware translation mechanism, that is distributed and realized on all nodes. It comprises of a forward/backward address translation scheme, tailored to the AXI protocol's inherent characteristics. An abstract representation of our approach is demonstrated in Figure 4.25. A wide global address, i.e. 41-bit in our prototype, should pass through the AXI interconnect. Address width itself is not an issue, as the AXI protocol, and any AXI compliant block, can support up to 128 bit wide addresses, or even higher. The forward/backward translation is deployed only around our central AXI interconnection block.

What we actually do, is to allow for an alternate addressing scheme, that exists in parallel and comes to life only in the reconfigurable part of a node. We are thus able to encompass the whole mechanism in a completely transparent manner, either

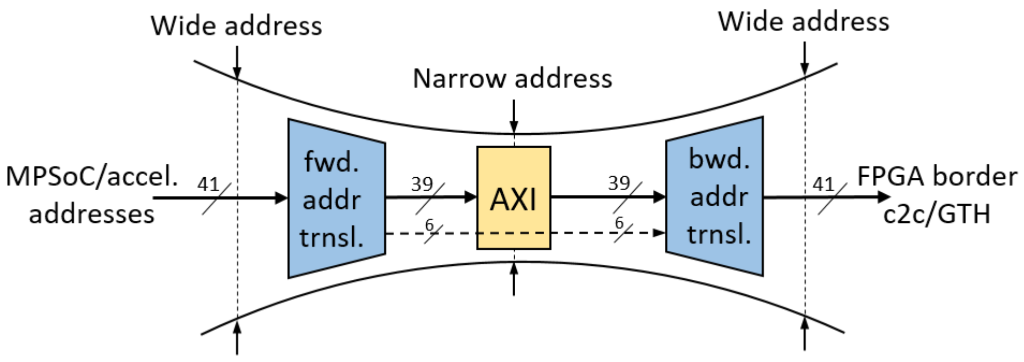


FIGURE 4.25: An abstract representation of a forward/backward AXI-aware address translation scheme

to the processor and any running software, or to any other peripheral accessing the global address space. These internal addresses consist of 39-bits. The lower 35 bits pertain to the 32 GB that comprise a Worker's address window and need not be translated. The remaining 4 MS bits are enough to differentiate among a local access and any of the 8 or 9 possible outgoing paths of figure 4.10 (3 for FPGAs of the current QFDB, 4 for accessing the directly connected QFDBs of the same cube, and 1 or 2 for targeting remote cubes). Each such possible target is mapped onto a distinct memory region. All these memory regions can now appear in a consecutive manner, and so become an easy match for the AXI interconnect. This is true since each node only has to address the next hop on the way to the destination node, using its translation map, and this is uniform in our architecture. This arrangement actually constitutes a *distributed routing scheme*, as a transaction advances across nodes. Any node needs to handle only a *single-step routing*, just resolving which is the way on to the next node.

Now to compose these 4 MS bits, a translation has to occur, that uses the 6 MS bits of the original 41-bit address and information for the current node's ID. Looking back to Figure 4.17 this takes place in the purple blocks of the datapath. To allow for uniformity of this translation mechanism across Workers, it gets orchestrated by a software-configurable, hardware translation table, also seen in purple in Figure 4.17. This translation mechanism actually has a twofold effect: not only does it help uniformity, but also helps so that the hardware translation blocks remain as simple as possible. To do so, the representation it holds actually comprises of a semi-precompiled address, so that only simple last steps have to be executed in hardware.

More details on reaching to an efficient translation table scheme to serve our objectives are presented in the following section, however its worth clarifying here the reverse translation. After a transaction is routed on the proper output of the AXI interconnect, Figure 4.25, the original address has to be reconstructed. In order to re-fit the original 6 MS bits of the address, they accompany the transaction through AXI's "user bits". These are configurable AXI bits that travel along with each AXI transaction, configurable to much larger sizes, and are depicted with dashed lines in Figure 4.25.

Both forward and backward address translation is carried out by specially developed hardware modules. In order to avoid bottlenecks, as several paths exercise these translation primitives, dedicated translation modules are replicated. As each transaction may travel a number of hops, and thus translations, before reaching its destination in an HPC-sized system, the translation modules are as simple, fast and scalable as possible. As a result they get implemented as pure combinational circuits, adding no extra clock cycles in any address path.

4.2.5.1 Details on how Cross-Chip Communication is Restrained by AXI IDs, and the Deployed Solution

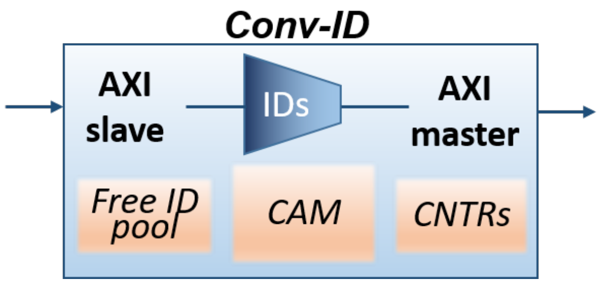


FIGURE 4.26: The Convert-ID (ConvID) logic block, properly translating AXI IDs

We already discussed on limitations having to do with deploying AXI interconnects for cross-chip communication. There is another important limitation on the same aspect, having to do with the AXI protocol itself, that involved a specially developed solution. This was initially part of the UNIMEM architecture implementation, however it now required special improvements in order to be deployed in our UNILOGIC approach, as well as contextual configuration and tuning, along with debug and upgrade to support the demanding accelerator traffic produced within the UNILOGIC platform.

This new entanglement that arises, is as follows. In the AXI protocol, each transaction is accompanied with a unique ID, which distinguishes it from any other transaction on the same path. As many AXI-compliant logic blocks connect to the same AXI interconnect infrastructure, the size (width) of this ID has to grow accordingly. This way it is able to differentiate transactions, which are initiated by a number of associated masters. When staying on-chip, this ID size can be easily bounded, as the number of AXI master blocks cannot grow enormously. However, this is not the case when we want to combindly consider multiple nodes, as should be the case in a large scale HPC system, consisting of thousands of nodes. In such a case, even if we could use an enormous size of ID bits in order to accommodate corresponding system sizes, the cyclic fashion of the master-slave transactions of the AXI protocol does not allow ID widths to converge. The AXI ID field of all slave blocks of an AXI interconnect should be of the same width, and each master's ID grows at every AXI interconnect crossing, depending on the master count. When multiple chips, i.e. nodes, get interconnected, each one contributes its enclosed AXI interconnect, and this leads the ID width definition into a kind of an endless incremental loop.

In order to resolve this, a special hardware module was developed, that engages in order to properly convert the IDs, so it is called convID (aka convert ID). The goal is to have a relatively small ID size even for complicated, multi-node systems, so it adopts in a way the approach of the Network Address Translation (NAT) used in standard networks. Thus a discrete convID block, abstractly depicted in Figure 4.26, has to intervene in any path that exits the current node, i.e. the FPGA. It has to change the ID, and narrow it down to a size that is compliant to the implemented architecture, and as such can by design be adopted by all nodes of the system. Subsequently, the transaction with the suitably converted ID will proceed to the network through the corresponding Chip-to-chip module.

The convID block hooks to the AXI datapath as a slave, with the appropriate ID width, dictated by the intra-chip architecture. After processing, it delivers the transaction on the master side, with a converted, narrow ID. We should note that narrow IDs can easily be handled by receivers (slaves) that accept wider ID, while the opposite case is that raises issues, as was presented above. In order for the convID to process the ID conversion, it owns a pool of unused IDs. These IDs are of a predetermined width and correspondingly a number of values. This is constant, configurable at design time, and is not directly depending on the number of nodes being added to the system. As a transaction enters the convID block, accompanied by a wide ID, the next available narrow ID of the pool is extracted, and is used to replace the wide one. The transaction is then forwarded with the newly appointed ID. The corresponding wide and narrow ID pair is stored in Content Addressable Memory (CAM). When the transaction response will come back, traveling in the opposite direction, the original, wide AXI ID, will have to accompany this response, in order to be properly identified by the receiver. A search in the CAM based on the narrow ID will efficiently retrieve the corresponding wide IP, and it can then be restored. When there are no pending transaction associated with a narrow ID, then it need to reenter the ID pool.

However, based on the AXI protocol, many ongoing transactions coming from the same initiator, will be coupled with the same ID. So, in order to accurately reestablish a free narrow ID keep, we need to keep track of the pending transaction count per ID. That is why an associated counter is deployed next to each ID pair, i.e. next to each CAM entry. Properly increasing/decreasing each counter, based on the AXI semantics, we can acquire precise knowledge of the ongoing transactions. Lastly, when for some reason, there exist two many ongoing transactions, and there is no available narrow ID in the pool, a forthcoming transaction will be stalled by proper AXI signaling, i.e. deactivating the ready signals.

We have up to now mentioned that the convID block is deployed on all the paths that lead to the network, i.e. leaving the chip boundaries. Likewise, it is deployed in all the paths that reach the current FPGA and lead to the processing system (PS) ports, i.e. targeting the local memory. These paths are traversed when a transaction designated the current FPGA as the final destination, and requests local data access. These PS ports are by vendor design build with a specific ID width (this equals to six for the Xilinx's MPSoC flavor we use). So these specific convID blocks, are entrusted with the task to narrow all incoming IDs down to this width, or narrower.

We should bare in mind that deploying a CAM introduces a notable amount of logic, with some parts of it growing superlinearly. Furthermore, CAM size is depending on the narrow ID width we choose, as the CAM entries grow quadratically ($entries = width^2$). In our current prototype, and after investigating on the usual requirements of hardware accelerators, as well as the usual spawning of hardware tasks by software applications, we ended up with a mixed configuration of convID blocks. Any block uses 18-bit wide (intra-FPGA) AXI IDs, i.e. in the slave side of the block of Figure 4.26. As for the master side, the convID blocks attached to memory ports use 4-bit wide (inter-FPGA) IDs. This is chosen as most usually there is an upper bound on paths that need to simultaneously drive the same PS port. Moreover, as we have already discussed, various incoming paths are selectively steered discrete PS ports, and likewise convID blocks. Another important aspect of the AXI protocol that allows for narrower IDs, is that in most cases, transaction initiated by the same master, obtain the same ID, which limits the number of active IDs, pertaining to outstanding transactions in the system. Lastly, the network attached convID blocks use 5-bit wide (inter-FPGA) IDs, as we want to allow an increased number of IDs. The paths involving links can act as intermediate hops inside the system, and may need to concentrate many incoming paths.

With these enhancements, the local FPGA interconnect remains oblivious of the chip-boundaries trespassing, and the AXI IDs can be effective, in a transparent manner, for both intra- and inter-FPGA AXI transactions.

4.2.5.2 Evolution and Complexities of the Internal Translation Scheme

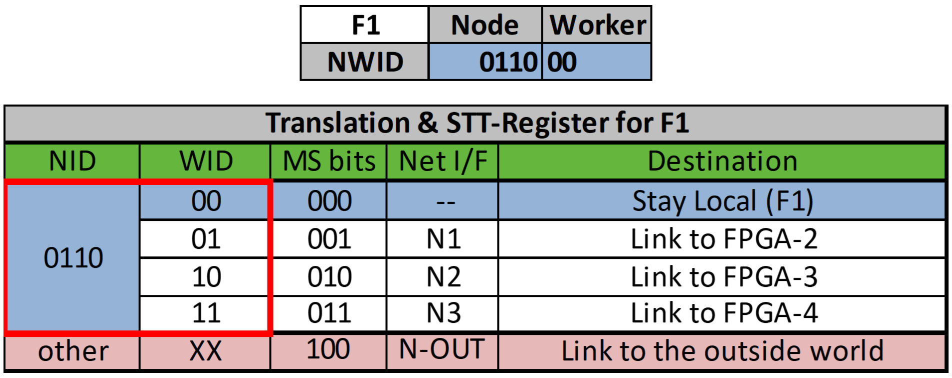
In this section we elaborate on how the translation tables where devised, in order to offer an easy to deploy, low-latency and low-resource solution. Each ongoing transaction in the system, targets a Worker's peripheral or memory. This Worker in turn is globally designated by both the Node-ID (NID) and Worker-ID (WID), which concatenated give the Node & Worker ID (NWID) value. We should think of this NWID value as a number designating a unique FPGA in the system. Based on the NWID, each transaction has to be navigated either to the proper block inside the same FPGA, targeting local resources, or towards a remote FPGA. In the latter case, it has to proceed through one of the FPGA's outgoing links. The associated routing has to be based on both the local Worker's (FPGA's) NWID, and the targeted NWID.

Devising a solution for HW-aided, topology aware, Node-ID designation

As a first step in the process, each FPGA has to gain its unique -topology aware- identifier, the NWID. This process has to be aided by hardware and done by the OS or any other software, e.g. bare metal, at startup. It is furthermore topology aware, in the sense that the underlying hardware components, i.e. the baseboard, offers NWIDs that are topology dependent. What is more, in order to support alternative topologies, these baseboard offered NWIDs can be configurable by SW running on a platform manager. The given NWID will become part of the Worker's translation table, which resides in the reconfigurable logic. We have considered various ways that allows each FPGA to gain its position on the system, encompassing awareness on the topology of the interconnection board. In our implementation, each baseboard is built to drive each FPGA with a number of bits through specific pins. In the FPGAs we deployed we drive the NWID through the General Purpose IO (GPIO) pins, or more specifically through the so called Extended Multiplexed I/O (EMIO) pins. These pins can be configured to pass through the FPGA's reconfigurable logic, and through a simple custom block we built reaches the Processing System's registers. Running software on the processor can then use this register to construct the proper translation table, as will be explained below. Elaborating on the NID bits reaching this register, each baseboard includes 3 bits to designate IDs from 0 to 7 for the 8 hosted QFDBs. Then extra bits are needed to designate the baseboard's number. In our implementation with two baseboards, one bit suffices, and this gives an aggregate of 31 = 4 bits, sufficient for the NID part of the NWID, i.e. to identify each of the 16 QFDBs. Both the first 3 bits, and the remaining baseboard-specific bits are created with a mechanism incorporated into the custom baseboard in a way that can be controlled by both static setup through jumpers, and dynamically through SW running on a programmable module such as a microZed or a Raspberry Pi.

<u>*Composing a proper translation table for the network-FPGA (F1)*</u>

This NID needs only enter the network-FPGA of a QFDB, also referred to as the F1 FPGA. The F1 can initially construct its own Static Translation Table (STT). It can then also proceed with constructing and remotely configuring the STTs of the other 3 FPGAs. In this way there is no need for any extra HW mechanism to inform these other three FPGAs of the NID their QFDB corresponds to and has acquired. Having constructed the appropriate STT values, the F1 FPGA is able to suitably configure each one remotely, as it is aware of each distinct link that leads to the corresponding remote FPGAs, i.e. F2, F3 and F4. And each such link is actually mapped to a corresponding region of the partitioned global address space (PGAS). Addressing a specific address within this region, ends up configuring the proper STT.



F1	Node	Worker
NWID	0110	00

Translation & STT-Register for F1				
NID	WID	MS bits	Net I/F	Destination
0110	00	000	--	Stay Local (F1)
	01	001	N1	Link to FPGA-2
	10	010	N2	Link to FPGA-3
	11	011	N3	Link to FPGA-4
other	XX	100	N-OUT	Link to the outside world

FIGURE 4.27: Static Translation Table (STT) snapshot for the network FPGA of a QFDB (F1)

As already mentioned, effort was made to construct a translation table that actually offers the related information in a partially pre-translated manner. This is crucial in order to simplify the hardware implementation, and as mentioned to achieve low-latency and minimal resource requirements. A snapshot of such a tailored STT is given in Figure 4.27. The specific STT presented corresponds to the F1 FPGA, and we will explain why. In the first column the current FPGA's NID is included, i.e. we are in the QFDB numbered 6 (0110). The second column includes an entry for each of the four possible Worker-IDs, i.e. FPGAs, inside a QFDB, i.e. from "00" to "11". Then, the third column includes the corresponding translated address for each of the four possible WIDs. Typically, only the MS bits need to be included, as low order bits remain untranslated. In this example, we can see in the first row and in blue color the WID = "00", which means that the current FPGA is the one numbered "0", i.e. we are in the F1 FPGA. It is worth reminding here that we need the 6 MS bits to be translated into 4 MS bits, however in this simple example only 3 resulting MS bits are needed.

Now we can think of a transaction, progressing on its way to the targeted logic, either memory or peripherals. This transaction carries a target address, designated by a valid point in the PGAS address space. It will now have to be driven through a forward address translation block, as the purple blocks of Figure 4.17. Each such – AXI-compliant – translation results to a new, FPGA-internal, corresponding target address. Looking to the blue-shaded row of Figure 4.27, the MS bits of the address corresponding to this row are equal to "0110-00", so it targets QFDB No.6 and FPGA No.0, i.e. F1. The third column specifies that this needs to be internally translated to "000", and this will steer it to the local FPGA peripherals, as noted in the last column. This is so, as such a transaction would target the local FPGA. As a next example, looking in the next row, we have the translation for a transaction that targets "0110-01". This needs to be translated to "001", and this way it will get routed to the F2 FPGA, through the "N1" network interface. The same goes for the next two rows of the table. The last row, in red, pertains to the case that the target address does not have a matching NID. This means that the target is outside the current Compute Node, i.e. QFDB, and will need to exit trough the outgoing link. This is achieved by translating the MS bits to "100". As this is a simple example, in order to incorporate our final topology, more "red" rows are needed, which steer to the proper outgoing link, depending on the targeted NID, and such a case will be presented below. Clearly, the addresses designated by the MS bits of the third column, should be correspondingly incorporated in the FPGA design, through the proper configuration of the central AXI interconnect address mapping.

Solving the uniformity issue of the translation scheme across FPGAs

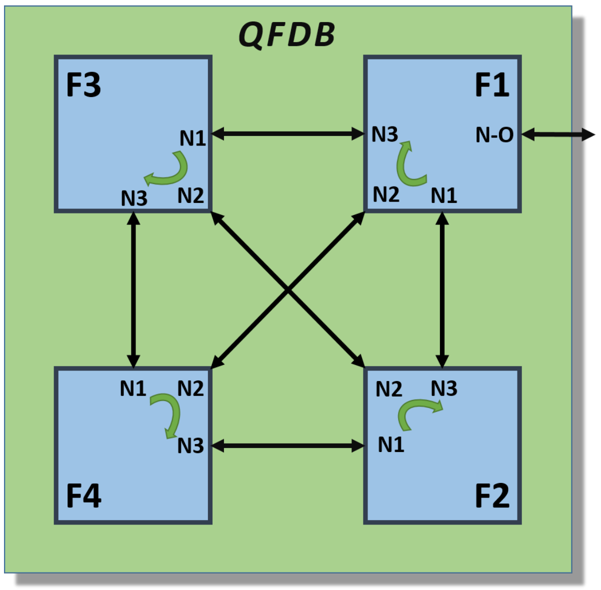
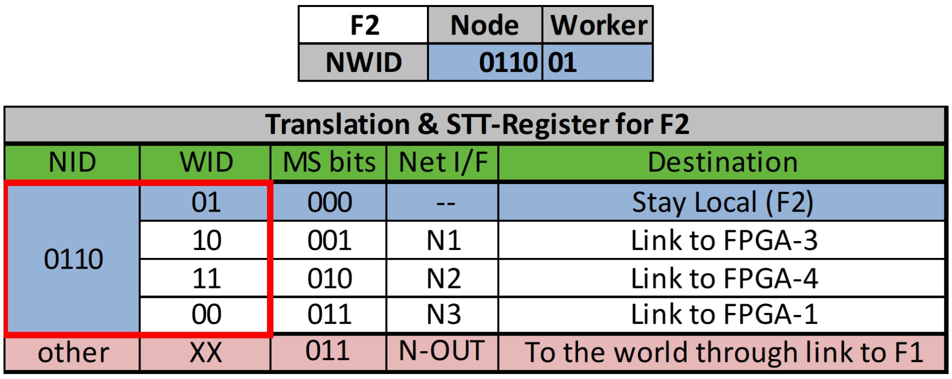


FIGURE 4.28: The cyclic notion for link selection of the network interfaces across the FPGAs of a QFDB offers h/w design uniformity

As the translation described above, has to be done for every FPGA of the QFDB, a problem emerges as with the uniformity of this approach. Unluckily, each FPGA on the QFDB incorporates a disparate transceiver designation. This means that a different address mapping has to be used for each of the four FPGAs.

Although this alone can be considered acceptable, it also leads to disparate translation to each FPGA, which in turn renders a uniformly pre-translated scheme non-applicable. So effort was devoted to solve this problem in a uniform manner. The proposed solution is to properly designate the links of each of the four FPGAs, i.e. the names of the 4th column of the STT. The qualified solution was that of assigning network interfaces, i.e. links, in a cyclic manner, as depicted in Figure 4.28. This cyclic designation of links, can allow for a uniformly devised translation mechanism across FPGAs. This also offers the basis for a more condensed, pre-translated hardware representation. And in turn this will also lead to less complicated logic for the hardware translation. This can be easier to comprehend by describing a second STT which is constructed under this approach, e.g. that of the second FPGA of a QFDB (F2). Such an STT is presented in Figure 4.29.



F2	Node	Worker
NWID	0110	01

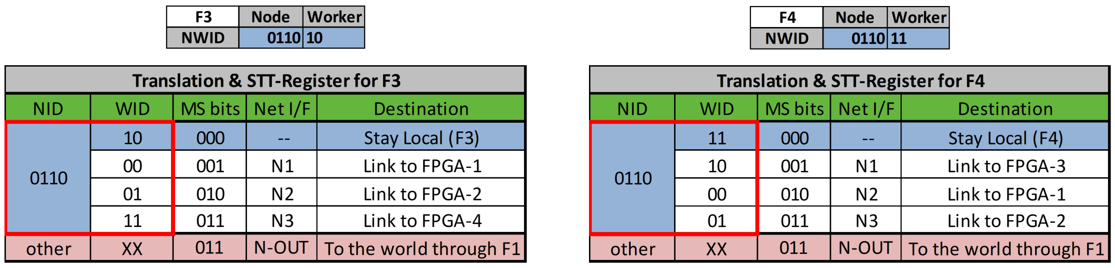
Translation & STT-Register for F2				
NID	WID	MS bits	Net I/F	Destination
0110	01	000	--	Stay Local (F2)
	10	001	N1	Link to FPGA-3
	11	010	N2	Link to FPGA-4
	00	011	N3	Link to FPGA-1
other	XX	011	N-OUT	To the world through link to F1

FIGURE 4.29: Static Translation Table (STT) snapshot of QFDB's FPGA 2 (F2). Comprises a pre-translated address notion, entailing the implemented topology

During the explanation of the devised translation scheme, it is helpful to have in mind both Figures 4.28 and 4.29. Looking at 4.29 we see a translation table similar to that of Figure 4.27. However, looking in more detail, we have now rearranged the table lines for each of the four WID column. This is done based on the proper permutation that leads to an identical 3rd column for all STTs of the four FPGAs. The "01" FPGA now appears first, as this is the address that should end up in the local resources, and can be seen highlighted in blue. Then the "10", i.e. the FPGA-3 is internally mapped to the subsequent address space "001", FPGA-4 to "010" and the network FPGA-1 to "011". This is actually a "rotated" representation of that in Figure 4.27, and adheres to what is dictated by the scheme of Figure 4.28. Additionally, any transaction that targets another

QFDB, again in the last row and shaded in red, now has to depart through the F1 FPGA. So any such address is now translated to "011", corresponding to the F1 FPGA as seen in the previous row of the table.

The proper permutation for the rows of all STTs can be extracted, as noted, based on Figure 4.28: each FPGA lists the three deployed link interfaces named N1, N2 and N3, but each of those has a different meaning in each FPGA, assigned in a cyclic manner. Thus for example N1 for F3 designates the link to F1, while N1 for F4 designates the link to F3. For all FPGAs however, we can think of N1 leading to the counterclockwise placed FPGA, N2 to the diagonally opposite FPGA, and N3 to the clockwise placed one. For purposes of better understanding, and as a complete reference we also include the STTs of F3 and F4 in 4.30



F3	Node	Worker
NWID	0110	10

Translation & STT-Register for F3				
NID	WID	MS bits	Net I/F	Destination
0110	10	000	--	Stay Local (F3)
	00	001	N1	Link to FPGA-1
	01	010	N2	Link to FPGA-2
	11	011	N3	Link to FPGA-4
other	XX	011	N-OUT	To the world through F1

(A) STT for F3

F4	Node	Worker
NWID	0110	11

Translation & STT-Register for F4				
NID	WID	MS bits	Net I/F	Destination
0110	11	000	--	Stay Local (F4)
	10	001	N1	Link to FPGA-3
	00	010	N2	Link to FPGA-1
	01	011	N3	Link to FPGA-2
other	XX	011	N-OUT	To the world through F1

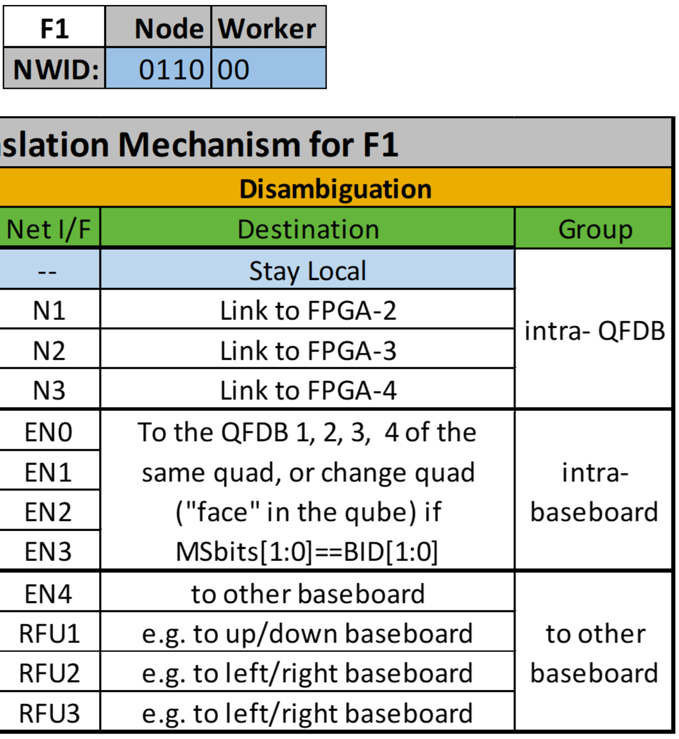
(B) STT for F4

FIGURE 4.30: Static Translation Table (STT) snapshots of QFDB's FPGA F3 and F4

In this way, the correspondingly permuted 2nd column, embodies all the needed topology designation. What is interesting, is that now we can omit the 3rd column in the HW translation table. By just including only what resides inside the red-outlined square, we give sufficient, pre-translated information for the actual translation to take place. For example, in any of the STTs, if the WID of an address matches the 1st row it always gets translated to "000", for the 2nd it translates to "001", and so on.

Incorporating the complete implemented topology in the translation table of each network FPGA

With the scheme we presented above, each FPGA can have a pre-translated STT, programmable by SW and allowing for an all-to-all interconnection inside the QFDB, thus utilizing the maximum underlying hardware connectivity of the Compute Node. Taking the next step with connectivity, the translation of the Network FPGA itself has to be enhanced, as we already mentioned, in order to include all the available routes of our implemented final topology. Since more outgoing paths have to be incorporated in the translated address, the enhanced scheme now needs to include 4 MS bits for this resulting address. Looking in Figure 4.31, the six NWID bits get translated to the four bits of the 3rd column. In blue shade, we see again the target address pertaining to the local FPGA. The designation of these four bits allows for the translated bits to get constructed by just concatenating "01" to the original address' WID, thus requiring no logic to compute. To achieve this, we had to encompass the proper AXI configuration and address mapping inside the FPGA, as was the case with the simplified F1 translation table we have seen before.



F1	Node	Worker
NWID:	0110	00

Translation Mechanism for F1					
Target Addr		New Addr	Disambiguation		
NID	WID	MS bits	Net I/F	Destination	Group
0110	00	0100	--	Stay Local	intra- QFDB
	01	0101	N1	Link to FPGA-2	
	10	0110	N2	Link to FPGA-3	
	11	0111	N3	Link to FPGA-4	
0-XXX	XX	10-00	EN0	To the QFDB 1, 2, 3, 4 of the same quad, or change quad ("face" in the qube) if MSbits[1:0]==BID[1:0]	intra-baseboard
		10-01	EN1		
		10-10	EN2		
		10-11	EN3		
other		11-00	EN4	to other baseboard	to other baseboard
		11-01	RFU1	e.g. to up/down baseboard	
		11-10	RFU2	e.g. to left/right baseboard	
		11-11	RFU3	e.g. to left/right baseboard	

FIGURE 4.31: The enhanced Static Translation Table (STT) for the network FPGA of a QFDB (F1), including the entire implemented topology

The remaining rows, which correspond to different outgoing paths and depend on topology, had to be constructed in a similarly efficient, easy to materialize in hardware, manner. Remember that each baseboard offers an all-to-all connectivity among the four QFDBs of each of the two quads, with such a quad presented before in Figure 4.6. Then these two quads get interconnected, edge to edge, resulting to the enriched cube connectivity we have seen in Figure 4.8. Furthermore, as we devised the baseboard through a co-design process, we have chosen to designate the appropriate transceivers to the appropriate QFDB positions in a advantageous manner. This is deployed in the hardware connections of the baseboard, and greatly simplifies the overlying translation hardware introduced in the QFDBs. Elaborating on this, and as briefly noted in the 4th column of Figure 4.31 labeled 'Destination', all the QFDBs of a quad have the same transceiver leading to the same FPGA. This means that e.g. "transceiver-1" leads to QFDB-1, "transceiver-2" leads to QFDB-2 and so on, and this stands for all four QFDBs. Additionally, each FPGA, e.g. FPGA-2, deploys the transceiver with the same number, i.e. "transceiver-2", to lead to the corresponding QFDB of the opposite quad.

Under this solution, the composition of the STT corresponding to the translation mechanism for the network FPGA can be accomplished in a more simple way. The second quad of rows for Figure 4.31 are in effect when the target Node is in the same baseboard. This is so if the MS bit of the local NID matches that of the target address. In a larger prototype, e.g. deploying more baseboards, the NID would grow proportionally, and accordingly more MS bits would be used. The translated address, aided by the interconnection co-design explained above, is composed by concatenating MS bits "10" to the two LS bits of the targeted Node, i.e. the 2 LS bits of the NID. As mentioned, the NID for this rows gives a match to the same baseboard, and thus this addressing need to just steer to the proper QFDB of the quad, or just change quad.

Finally, the last four rows are used when a different baseboard is targeted. In this case we concatenate "11" to the 2 LS bits of NID. In our prototype only the first of these last rows is valid, leading to the topology of Figure 4.9. This is why the next rows are marked as Reserved for Future Use (RFU). If we deployed a bigger prototype, the second of these rows could be used to serve a topology as this of Figure 4.10. Furthermore, using the high-speed multiplexers residing in the baseboard as mentioned in section 4.1.4, thus designating more baseboard-outgoing links, fancier topologies could be implemented, e.g. realizing a 2-D torus of Cubes using the up, down, left and right address translations, as indicated in the last two rows of the table.

Importantly, the aforementioned straightforward composition leads to a correspondingly low-resource and low-latency hardware translation mechanism. As a matter of fact, we were able to include all the intervening translation blocks without any registered logic, thus not adding any clock cycles to the datapaths they interpose. We furthermore thoroughly measured the clock rates achievable with and without the UNILOGIC translation blocks and we observed no change to the critical paths. In that manner, we can in a sense consider the resulting scheme as a zero latency translation.

Optimizing throughput to memory by deploying four parallel, globally accessible Processing-System ports

As we moved on with our investigation on the efficiency of the UNILOGIC approach, a subsequent issue emerged that had to do with memory bandwidth. Although seemingly unrelated with the AXI interconnect, the solution is closely related to the AXI addressing. The problem we had to solve is as follows. The DDR memory, paired with each MPSoC of the QFDB, deploys a 64-bit wide data bus. We have managed to clock this interface at 2133 MHz DDR frequency (i.e. with a 1066.66 MHz clock). At this clock rate, the DDR memory should be able to provide an ideal peak throughput of about $2133 \times 64 \simeq 136$ Gbits per second, or equally about 17 GBytes per second. However, the circuitry inside the Processing System (PS) of the MPSoC, within which the DDR controller is included, allowed us to measure a peak throughput close to 9 GBytes per second, or in the best case 10 GBytes per second. This result is indeed in par with the maximum achievable throughput reported by most recent publications such as [76], while it is highly improved compared to the other evaluation results reported such as [8], which does not incorporate burst transactions. In order to reach this peak measured throughput, we had to deploy four slave ports of the PS, all running in parallel and accessing the DDR. The PS offers 6 such ports in total, however not all are internally independent (i.e. two pairs go though the same multiplexers). We have performed various low lever benchmarks in order to properly decide which configuration produces the optimal results. Based on this, and on our targeted FPGA architecture, we ended up with selecting 4 of these ports, which give the most efficient, combined configuration.

The challenge we had to confront next was to encompass these four ports to our design. When we want to run accelerators only locally, i.e. allow them to access only local memory, then each of these accelerators can get hooked to each of the four PS ports. We have gone even further on such explorations during our research, by even adding up to 12 accelerators, organized in six pairs and with each pair sharing one of the total 6 PS ports. However, when we opt to allow accelerators

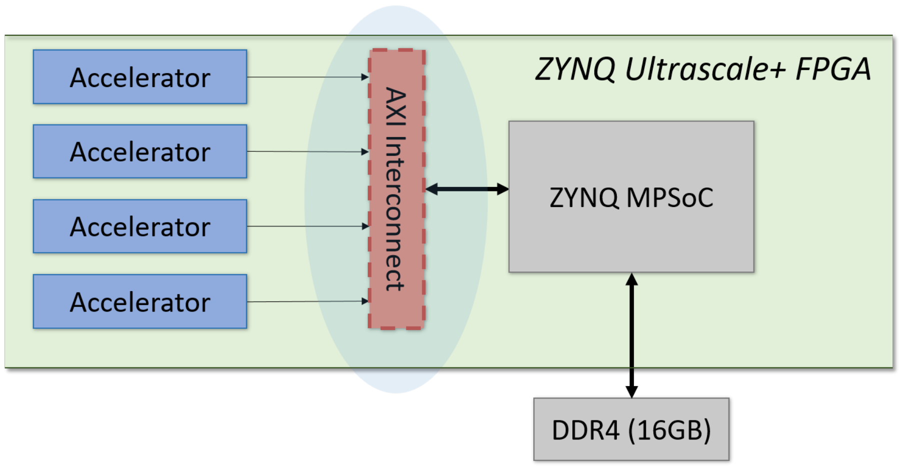


FIGURE 4.32: A simplified portion of the FPGA block diagram, highlighting that in order to have a global all to all resource access, accelerators just as any other peripheral, have to pass through the central AXI interconnect. Thus the get restricted to a single point of entry towards the memory

to access any memory in the system, or symmetrically allow the memory ports to be accessible by any entity in the system, the AXI interconnect has to intervene between the PS and the entity, i.e. processors, accelerators, network ports etc. This was seen in Figure 4.17, but can be better understood through the highly simplified view of the block diagram in Figure 4.32, showing only the accelerators, that now along with any other peripheral have to traverse the common central AXI interconnect. In this case, the AXI is not capable to include multiple distinct PS ports to its address map. This is so, as all the available master PS ports map to the same regions, including the actual memory addresses, as was presented in Figure 4.22. So the AXI interconnect has no way to differentiate memory accesses targeting a specific memory portion, has no justification to select among various peripherals with the same mapping, and thus reasonably disallows multiple PS ports to get connected. Under this restriction, a single path to the memory has a limited throughput: the maximum allowed datapath width of each PS port is 128 bits, and e.g. at a 200 MHz clocked design would reach a peak throughput of $200 \times 128 = 25.6$ Gbits per second, or equally about 3.2 GBytes per second. As understood, this causes a hard bottleneck, that would greatly impoverish our implementation.

What we propose and implemented to resolve this restriction, is adding a custom built address dispersal block between the forward translation blocks and the

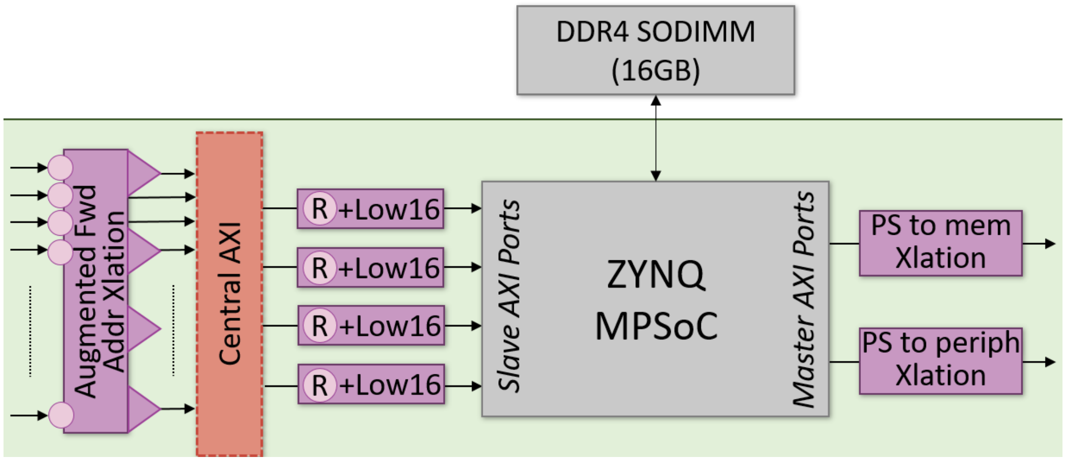


Figure 4.33: A detailed portion of the abstract FPGA block diagram, showing the deployment of four slave PS ports in parallel, as well as both master ports, and the necessary augmented Address Translation blocks

AXI interconnect. This address dispersal block, alters any transaction that targets memory. What it does is to translate each memory address into one of 4 possible discrete regions, of the same size as the actual memory. This way, as the altered transaction enters the AXI interconnect, it now targets one of four corresponding discrete blocks. These blocks are custom built and are AXI compliant in order to be address mapped. Each such block is in turn assigned with the task to reverse translate transaction addresses, from discrete regions into the region that maps to the original memory addresses. They are mapped as slaves to the AXI interconnect, at each of the different address regions than that of the memory, and of the same size. Then each such block is hooked as a master to each of the PS ports, and has to transpose each access into the actual memory region. This memory region pertains to the lower 16 GB of the 32 GB region assigned per Worker.

In our final implemented version we moved on with integrating this block with the overall translation scheme, with a detailed portion of the block diagram presented in Figure 4.33. This Figure actually corresponds to an expanded representation of the upper left part of Figure 4.17. The augmented translation block depicted, constitutes a specialization of the generic translation blocks we have referred to before, and is deployed solely at the paths that are allows to route to the memory. The purpose of this integration with the translation block, is to allow for better optimizations inside the mixed block logic. This way we were able to still devise a resulting hardware translation mechanism that essentially introduces no added latency.

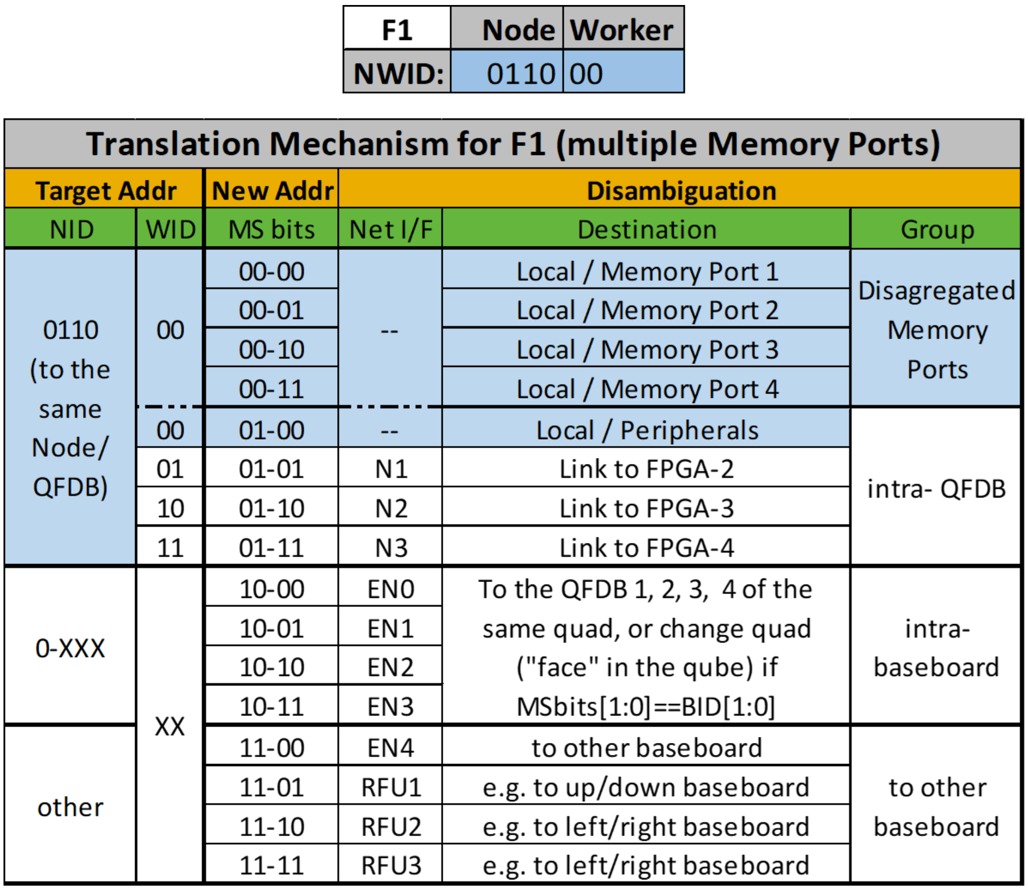
Now as seen at the leftmost part of Figure 4.33, any number of entities that want to access the local memory, will have to pass through the augmented address translation. Then, any of the access that targets memory will exit the translation block with an new target address, and will enter the AXI interconnect. The memory address dispersal blocks are abstractly represented by a circle in the entry point of each incoming path to the translation. Then, exiting the AXI interconnect, any memory transaction will have been routed to one of the four blocks leading to the PS ports. Any other access that does not target memory, will exit the AXI from different ports, not shown in this explanatory view. These blocks will reverse translate the address to the original memory address, thus designated by "R" inside the block, and will thus properly deliver the transaction to the DDR memory controller.

As for the "Low16" designation in this block, we should have in mind that the lower 16 GB of the address space do not actually comply with the Xilinx's MPSoC architecture. Adversely, this is split in two regions of 2 GB and 14 GB, and so the "Low16" logic has to properly restructure the address, as was presented in section 4.2.4. To better allow logic optimizations, we once again merged the two address handling blocks, so now the "Low16" part is also annotated with the "R" part of the reverse translation, i.e. the capability to transpose any of the four equal –dispersed– address ranges, back to the originally targeted 16 GB of DDR memory. It is important to also note that possible entities initiating memory targeting transactions include a local processor or a local accelerator and also a network interface that likewise delivers a transaction equally initiated by a remote processor or accelerator. So before entering a PS port, the 'AXI-ID-conversion' function need to also take effect, so a "conv-ID" block as presented in section 4.2.5.1 need to also intervene in this path to memory, not depicted here for simplification.

An important last step to be explained, is how to advantageously take care of the assignment of memory accessing paths to PS ports, i.e. to on of the four address dispersal blocks. As explained, each one has a distinct address mapping. We have to originally alter the memory targeted transactions of each incoming path to one of those regions. As a first distribution, we have chosen 1) to appoint one such block per local accelerator block. This way, when all four accelerators run in parallel, they will gain maximum memory throughput. This is important as such a case of stressing local accelerator is expected to be an often one. Then, as transactions come from the network interfaces, we have appointed, each such interface to one of the four blocks. However, instead of simply performing the assignment in an oblivious cyclic fashion, we have rather chosen to separately assign network interfaces that are expected to operate simultaneously, and thus produce concurrent accesses to memory. So 2) FPGAs of the same QFDB, which are regarded as first class neighbors, are assigned to different blocks. Usual multi-accelerator scenarios are expected to mostly deploy the intra-QFDB neighboring Workers, i.e. FPGAs, and make use of a data sets residing in either a single DDR module, or other modules of the local QFDB. Similarly, on the opposite scale, we 3) assigned interfaces that lead to large subsets of nodes in disjoint PS ports. This separation was performed with the implemented topology in mind, and with the routing priority implemented, and will be explained below.

We have also formulated a further step, with designing and implementing a version of the dispersal blocks, that alternates the PS port assignment in a round robin fashion. That is, as a series of transactions enter the FPGA, each one is cyclically assigned to a different block. This would offer the maximum versatility. Each path can dynamically associate to different ports, resulting to the aggregate traffic of all incoming paths spread equally among the memory-targeting ports. This would still stand, no matter which or how many of these paths are injecting the heaviest traffic. Unfortunately, this cannot comply with the AXI protocol, as AXI does not support out of order completion of transactions. And in a large scale system, dispersing a series of transactions to different targets, would lead to out of order responses, which cannot be carried out by the AXI protocol and likewise by the central AXI interconnect.

Finally, in this detailed portion of the block diagram, surrounding the Processing System, we have also included the two master ports, already mentioned firstly in section 4.2.1. As discussed, the main master ports are used to globally access memory and peripherals, e.g. accelerator controllers, respectively. The memory appointed block bears the task to implement the two-stages addressing for global memory. The peripheral appointed block has to properly process a corresponding, SW-originating, global peripheral address. Both mechanisms were described in section 4.2.4 and manage to compile narrower addresses generated in the PS, into UNILOGIC-compliant, wider global addresses.



F1	Node	Worker
NWID:	0110	00

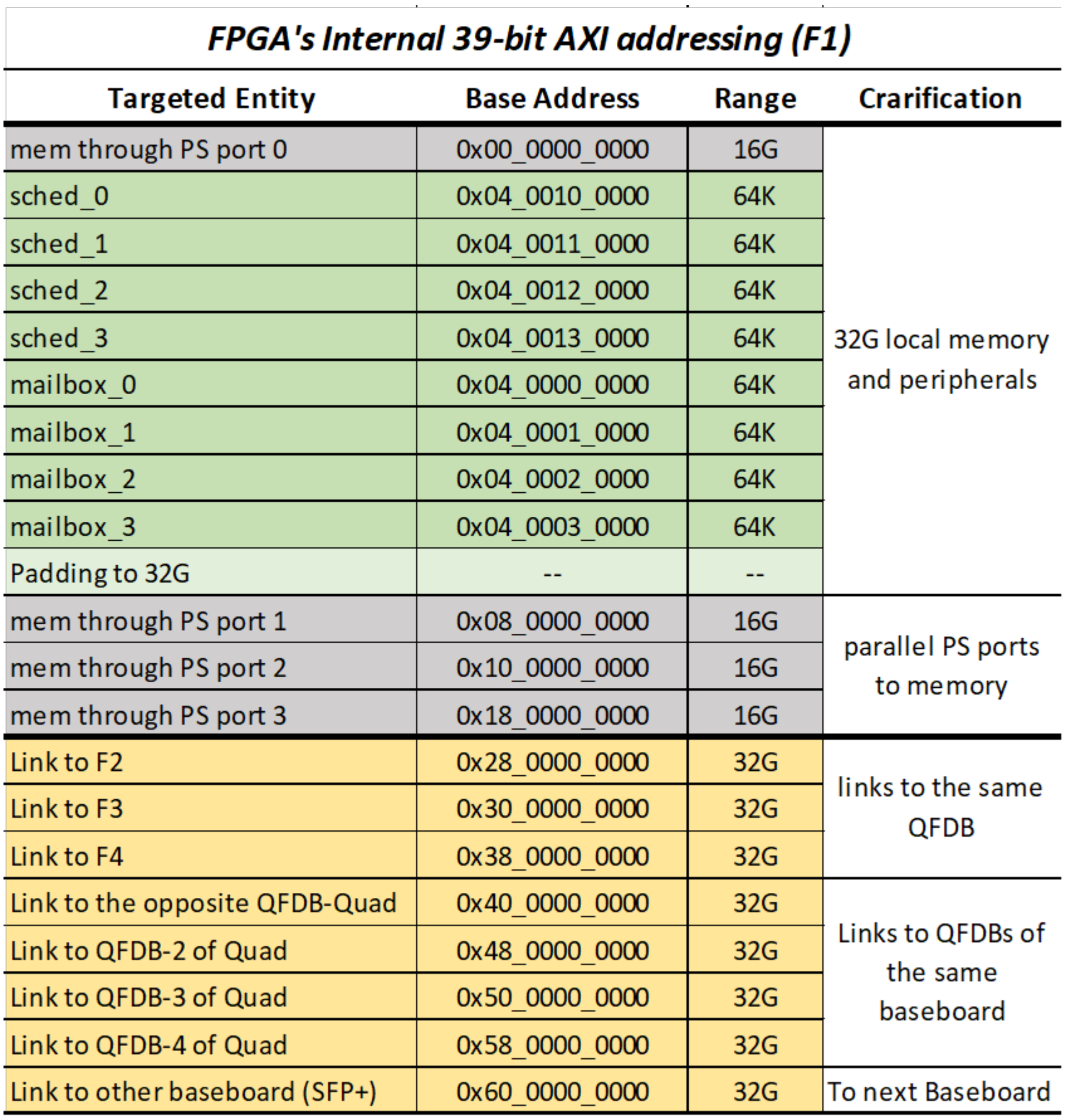
Translation Mechanism for F1 (multiple Memory Ports)					
Target Addr		New Addr	Disambiguation		
NID	WID	MS bits	Net I/F	Destination	Group
0110 (to the same Node/ QFDB)	00	00-00	--	Local / Memory Port 1	Disaggregated Memory Ports
		00-01		Local / Memory Port 2	
		00-10		Local / Memory Port 3	
		00-11		Local / Memory Port 4	
	00	01-00	--	Local / Peripherals	intra- QFDB
	01	01-01	N1	Link to FPGA-2	
	10	01-10	N2	Link to FPGA-3	
	11	01-11	N3	Link to FPGA-4	
0-XXX	XX	10-00	EN0	To the QFDB 1, 2, 3, 4 of the same quad, or change quad ("face" in the qube) if MSbits[1:0]==BID[1:0]	intra-baseboard
		10-01	EN1		
		10-10	EN2		
		10-11	EN3		
other		11-00	EN4	to other baseboard	to other baseboard
		11-01	RFU1	e.g. to up/down baseboard	
		11-10	RFU2	e.g. to left/right baseboard	
		11-11	RFU3	e.g. to left/right baseboard	

Figure 4.34: The network FPGA STT, annotated with proper translation for allowing multiple ports to memory

Incorporating this final annotation of the UNILOGIC implementation for multiple memory ports, adds to the internal address representation, i.e. the Static Translation Table, and brings it to the final form, now presented in Figure 4.34. Once again, the blue shaded part corresponds to local accesses, which however has grown lengthier. It now incorporates four identical regions for the local memory. Targeting each of this regions, will actually lead to one of the four PS ports as discussed. The proper forward translating logic is configured statically, as also discussed, in order to steer transactions coming from each of the possible paths, on to the proper port. Then, once reaching just before this port, the proper logic will again reverse translate the addresses to the originally targeted memory region, actually residing in the lower section, i.e. the first blue shaded row of the STT. Again we aimed to allow for the translated MS bits to be composed through the simplest possible circuitry, supported in parallel by properly configuring the AXI interconnect address mapping.

The final internal central-AXI address mapping for the Network FPGAs

For a better insight and as a thorough reference, we present at this point the resulting central-AXI address mapping for the network Worker of each compute Node, i.e. the F1 FPGA of each QFDB. This address mapping, as presented in Figure 4.35, was realized in a side to side co-design process with all the incorporated optimizations and hardware simplification methods already described above. This results in a tailor made intra-FPGA, AXI-compliant address mapping, that perfectly suits the progressive, forward-backward address translation scheme. The internal representation, as described, includes 39 bits for our implementation, and can be easily configured to support implementations of highly increased size.



FPGA's Internal 39-bit AXI addressing (F1)

Targeted Entity	Base Address	Range	Crarification
mem through PS port 0	0x00_0000_0000	16G	32G local memory and peripherals
sched_0	0x04_0010_0000	64K	
sched_1	0x04_0011_0000	64K	
sched_2	0x04_0012_0000	64K	
sched_3	0x04_0013_0000	64K	
mailbox_0	0x04_0000_0000	64K	
mailbox_1	0x04_0001_0000	64K	
mailbox_2	0x04_0002_0000	64K	
mailbox_3	0x04_0003_0000	64K	
Padding to 32G	--	--	
mem through PS port 1	0x08_0000_0000	16G	parallel PS ports to memory
mem through PS port 2	0x10_0000_0000	16G	
mem through PS port 3	0x18_0000_0000	16G	
Link to F2	0x28_0000_0000	32G	links to the same QFDB
Link to F3	0x30_0000_0000	32G	
Link to F4	0x38_0000_0000	32G	
Link to the opposite QFDB-Quad	0x40_0000_0000	32G	Links to QFDBs of the same baseboard
Link to QFDB-2 of Quad	0x48_0000_0000	32G	
Link to QFDB-3 of Quad	0x50_0000_0000	32G	
Link to QFDB-4 of Quad	0x58_0000_0000	32G	
Link to other baseboard (SFP+)	0x60_0000_0000	32G	To next Baseboard

FIGURE 4.35: The network FPGA address mapping for the central AXI interconnect, based on the HW simplification objective

Each Worker, i.e. FPGA, as mentioned above exposes 32 GB of address space to the system's PGAS. 16 GB for a full access to the existing DDR memory, and then extra address space for the peripherals, complemented by "RFU" padding to properly align at 32 GB boundaries. The local memory region can be seen in the first row of Figure 4.35, mapped to the lower 16 GB and shaded in grey. Then, in the green shaded rows, we can see the main peripherals targeted, i.e. the schedulers and mailboxes. Each scheduler-mailbox pair actually constitutes an accelerator controller. Four such pairs are shown here, while depending on implementation, this can be tuned to include more or less such controllers. Proper padding follows this region, and aligns it to 32 GB, as seen in the bottom green-shaded row and clarified by the rightmost column. A useful detail to notice, is that the "0x04..." addresses of the peripheral mapping, assign the 35th address bit to 1, therefore designating a peripheral, as dictated by the addressing scheme presented in Table 4.1. Conveniently, the lower 35 address bits remain untranslated, and get concatenated with the forward-translated 4 bits, to form the 39 bit address. After properly routed inside the AXI interconnect, a transaction will undergo backwards address translation which will restore the original 6 MS address bits, and will then be forwarded on to the next destination in its path.

The following rows in Figure 4.35, shaded in grey, include three "mirror" copies of the 16 GB DDR memory address map. As explained, each one will lead to a different port of the PL-to-PS interface, and then on to the DDR controller, in order to avoid single port congestion. This way, following the path of a transaction, any master-sided module that wants to access memory is configured to target one of these regions. Custom hardware just before each PS port will reversely translate any such region to the only existing one, that recorded in the first row.

Moving on with the address mapping table, next come the yellow shaded rows, which designate all the outgoing paths, i.e. the paths that lead outside the chip through the network interfaces. These pertain to the links that are driven by the Xilinx's GTH multi-gigabit transceivers. As the translation happen partly in any of the network FPGAs traversed, it can be as distributed, and takes place progressively. Due to this progressive address translation, all address regions of the complete platform need not appear in this mapping. Only the next hop, i.e. next node, needs to appear, and this aspect greatly enhances scalability. Elaborating on this, and as clarified in the rightmost column, the first 3 yellow rows designate the network interfaces to each of the other three FPGA residing in the same Compute Node, i.e. QFDB. Then, having in mind the enriched cube topology offered to the hosted QFDBs by the baseboard, the next four rows give the mapping for addresses targeting the same baseboard. Each QFDB is a vertex in one of the two QFDB quads, or likewise to the QFDB cube. Proper hardware link assignment allows to target any vertex in a uniform way from any other vertex. One region is assigned for each vertex, while targeting the "identity vertex" will lead to the same vertex of the opposite quad. In other words, in the QFDB-1 of a quad, the first of these four regions leads to the opposite quad's QFDB-5, as designated in the table. Likewise, in QFDB-2 of a quad, the second of these four regions leads to the opposite quad's QFDB-6, etc. This designation comes naturally due to the devised scheme, and does not require any specific configuration for discrete QFDBs.

Subsequently, as designated in the last row, if the address matches none of the local baseboard's eight vertices, i.e. QFDBs, the transaction has to be routed on to the next baseboard. In our implementation, only one such other baseboard exists, however more baseboards can be added, connected either in a ring topology, or even realizing more dense topologies such as 2D-tori. Since the baseboard-outgoing link leads to an SFP+ connector, depending on how the cables are connected, we can reach a different vertex of the opposite cube, i.e. the one on the opposite baseboard. We tried various connections to verify our architecture and implementation, while in most of the evaluated cases we used the most inherent one, that is connecting a vertex to the same vertex of the opposite cube.

A interesting matter to describe at this point, is that when a transaction has to be routed to a distant QFDB, a lot of different paths exist. Depending on the precedence designated by the translation blocks, a transaction can be maneuvered to follow any of the –mainly– shortest paths to the destination. For example, in our tests we usually configure the translation blocks to firstly direct for a baseboard crossing, i.e. change cube, then quad crossing, i.e. change quad inside a cube, and then QFDB, i.e. one of four inside the quad. This method can also intrinsically incorporate a method to split traffic, as we can differentiate the path between two nodes designated as an initiator and a target. This happens as one path can be followed for the request, and a different –symmetrical– one for the response. A request that leaves the initiator travels on to the next cube, then quad, then QFDB, while the responds travels the opposite way back. However if this target needs to act as a initiator, e.g. serving a different application, and issue a transaction to the other node, then it would follow a different route, as it would symmetrically first change cube etc. It would actually reach the other node through a path that conceivably creates a circle when coupled with the previously followed path. Nonetheless, as the address translation blocks are designed to be easily configurable, proper configuration can conveniently alter this steering precedence. It can be done both statically or even at runtime, provided the process followed does not violate the AXI protocol specifications.

Finally it should also be mentioned that the F2, F3, and F4 routing, i.e. the routing for the other three FPGAs of each QFDB, remains simplified. So the translation tables, address translation blocks and corresponding address mappings remain much more simple, and actually pertain to what was presented through our initial analysis of the STTs in the beginning of this section and around Figures 4.29 and 4.30. This is so, as in all cases when inside these FPGAs, an originating transaction that has to leave the current QFDB, only needs to be routed to the Network (F1) FPGA. This Network FPGA will then be responsible for properly routing the transaction to its actual destination.

4.2.5.3 Exploration on reducing AXI interconnect resources

As has been already described, the UNILOGIC architecture needs to be composed of two main logic segments: the static part implementing the interconnection, routing, partial reconfiguration support, as well as the accelerator controllers, and the dynamic part, incorporating the accelerators, either configured on power up, or by allowing large placeholders, i.e. partial reconfiguration slots, that can be dynamically reconfigured at run time. Thus, a main objective while implementing the UNILOGIC architecture, and to better prove that it constitutes an effective approach, is to take care in order to consume as less FPGA resources for the static part as possible. This way, the hardware logic can spare the highest possible percentage of FPGA estate for the accelerators themselves. In this section we will explain our efforts and the methods followed and evaluated to economize resources spent for the central AXI interconnect. This logic block accounts for a large amount of the static part, so it constitutes a main target in order to achieve logic reduction. As for the rest of the static part, we have already mentioned various techniques that were incorporated, through which we also economized that portion of the logic as well. Then, in a following, summarising section we will also present a brief recap of the main resulting resource overheads for all four FPGAs of the QFDB.

Table 4.4: F2, F3 & F4 FPGA resources consumed for various AXI interconnect approaches

Strategy (Full topology, 4 slots)	**Total Static**	**Central AXI**
full topology, 4 slots (default)	15.27%	7.87%
default with **2-Level AXI**	8.30%	2.40%
2-Level AXI **with Regs**	8.70%	2.80%
2-Level AXI with Regs, **reconf**, Synth:PerfOpt	10.86%	3.46%
2-Level AXI with Regs, reconf, **Synth:Default**	9.43%	2.78%
full topology, 4 slots, Regs, 4 PS-ports	21.14%	11.91%
2-Level AXI with Regs, reconf, Synth:Default, **4 PS-ports**	**12.05**%	4.62%

We will start initially with the UNILOGIC implementation of the three FPGAs excluding the network one, i.e. the F2, F3 and F4, which have an almost identical design. A resume of the main approaches and optimization strategies we followed will be explained through Table 4.4. Many other variations have been implemented and evaluated, however not presenting any significant, additional information at this level of analysis. At first we can see the result for a full topology for these FPGAs, i.e. giving an all to all connectivity among the four QFDB's FPGA. In parallel the design allows allowing global access to and from any of four accelerator slots. This results in an AXI interconnect that spends about 8% of FPGA resources, with the aggregate static design consuming about 15%. The critical FPGA resource mostly spent in all these cases is Look Up Tables (LUTs), so all the percentages reported pertain to LUTs. Furthermore, all the important resources will also be later presented in brief.

The first and most important improvement we deployed, was *splitting the central AXI interconnect into a hierarchical AXI interconnect approach*. An insight on this can be provided though Figure 4.36, which presents a snapshot for a split-AXI strategy. Importantly, this was similarly implemented on all four FPGAs of the QFDB. Splitting the AXI into a hierarchical interconnect greatly reduces the size of the main crossbar, which in both cases resides in the Level-1 interconnect. Prior to splitting, the central crossbar becomes very large, and consequently less efficient it terms of clock speed. By splitting the central AXI, one has to be cautious, as to properly arrange and configure the L2 interconnects. For instance, in Figure 4.36 the accelerator controllers, i.e. mailboxes and virtualization schedulers, are grouped together, driven by a single master port of L1 AXI, as these receive only configurations commands and do not produce a lot of traffic. On the contrary, the four masters driving the four lines in the center, correspond to the four memory ports, and should thus remain autonomous. Subsequently, referring to the interconnect driving to the network interfaces, a single AXI is presented in this example, however it usually gets split into two or more smaller interconnects. This optimization, as seen in the 2nd row of Table 4.4 greatly reduces the resources spent, which now drop to a mere 2.4% for the hierarchical AXI interconnect, and 8.3% in total.

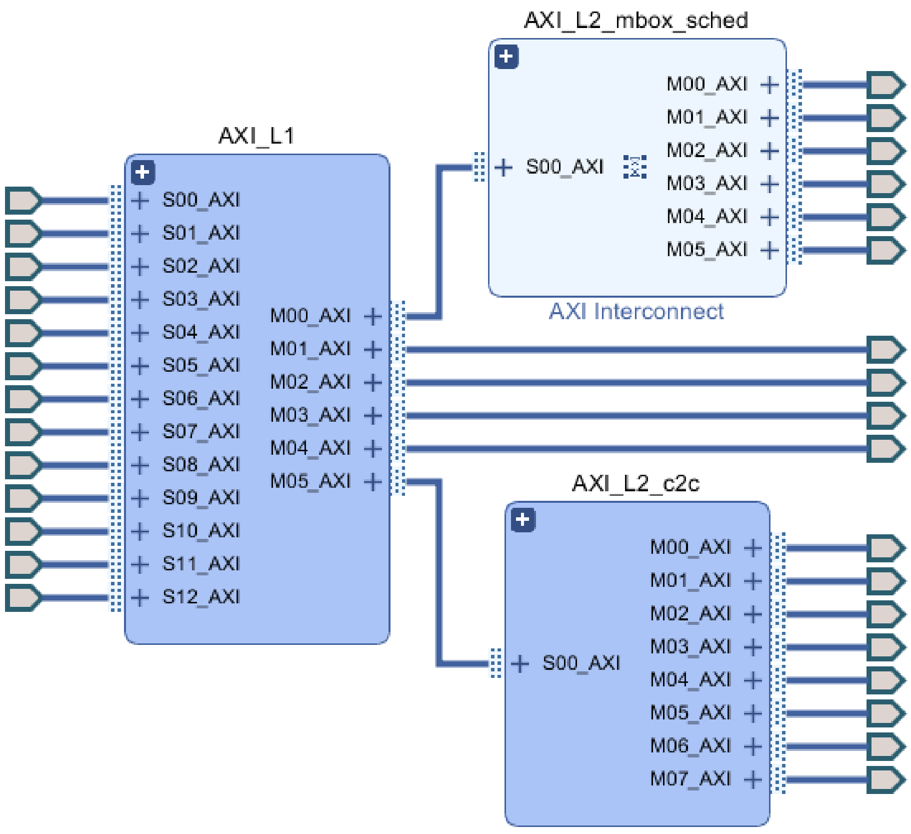


FIGURE 4.36: The hierarchical central AXI interconnect, greatly reducing FPGA resource requirements

Moving on based on this design, we added registers at the input and output ports of the AXI interconnects, which greatly improves timing closure, i.e. helps reach a higher targeted clock rate. In parallel we reverted to the "unmanaged" AXI configuration, which allows the designer to manually raise the supported outstanding transactions. However, both these optimization were not applied in a blindfolded fashion. As the AXI is now split in two levels, we are able to *deploy tailored optimization per AXI port and related path*. Particularly, we can add registers only to the input and output ports that actually engage in critical paths, thus avoiding unneeded overhead. Also, with the crossbars now being smaller, timing closure is further eased. Furthermore, continuing our investigation on selective AXI configuration within the hierarchical approach, we see that a high number of outstanding requests is not needed for all paths. We can economize on the configuration related ports, as the ones driving the accelerator controllers, or even the network interfacing C2C modules, while we can opt for the maximum AXI allowance on the memory related ones. We could even select some of the AXI blocks in the hierarchy to remain managed, i.e. their default initial state, as for example in the case of the AXI 2nd-level block leading to the accelerator controllers. This is so in the example of Figure 4.36, with the block on the top right that drives the accelerator controllers configured in its default state, and shaded for that purpose in lighter blue. Under this approach, i.e. adding the selectively registered AXI together with fine-tuned multiple outstanding support, only raises consumed resources by 0.4%, as seen in the 3rd row of Table 4.4.

Then we moved on to adding logic that fully supports the partial reconfiguration process for the accelerator slots deployed. This includes additions such as a configurable clock wizard in order to allow for separate clocks between the reconfiguration slots and the static logic. Also the Xilinx's hardware Internal Configuration Access Port (ICAP) block, which allows for partial bitstreams to be driven to reconfiguration slots. Importantly, deploying this block within the UNILOGIC architecture, and driven through the AXI interconnect, allows us to perform both local and remote partial reconfiguration. Furthermore, required reconfiguration logic decouplers were added per slot. These additions increase the static logic, and slightly the AXI interconnect itself due to the addition of the ICAP controller as a slave. In the first stages that we incorporated reconfiguration, we had tested various *alternatives to the Synthesis strategy*. Up to now, as the FPGA utilization, including the accelerators was not that high, we used the Vivado tool's "Performance Optimization" directive in the Synthesis flow. Under this strategy, the resulting static resources get raised to 10.86% and 3.46% for the total static and the AXI respectively, as seen in the 4th row of Table 4.4. This is so as the tool flow increases logic in favor of timing closure. However, with all our aforementioned optimizations deployed, timing closure gets greatly aided, and we could now opt for less stressing Synthesis directives. Even with the "Default" directive and as the tool does not struggle to achieve timing, no extra optimization-related logic gets introduced, and the logic percentages now fall to 9.43% and 2.78% as seen in the 5th row of Table 4.4.

Finally on the last row, we can see the resources required for one of our latest and "qualified" approaches, i.e. an approach that was deployed in our final prototype. Using the adequate "Default" Synthesis directive, we could now even add all four independent paths to the DDR memory. Adding these paths does not hinder timing closure. It increases the size of the level-1 AXI, but still remains within a modest overhead range, while offering the desired efficiency. The total static logic overhead now reaches at about 12% of the FPGA, with the hierarchical AXI contributing 4.62%. On second to last row, we report on a version that gives a metric of how large the static resources can grow in the absence of the examined alternative schemes. We have included four PS port support for memory access, which enlarges the AXI internal crossbar, and also added input and output registers to easier achieve timing closure, along with the support for multiple outstanding transactions. Deploying none of the specially examined optimizations, the AXI alone grows to about 12% of FPGA resources, with the total static overhead reaching about 21%. What is more, in this configuration the tool fails to achieve timing closure for the requested 200 MHz clock frequency.

Moving on to the Network FPGA (F1), we have also investigated on various schemes as can be seen through Table 4.5. In F1 the AXI crossbar becomes much larger, as it gathers more network interfaces to interconnect. These are needed in order to support the required outgoing links, and consequently the implemented topology. One added optimization we incorporated on all the FPGA designs, but becomes mostly important to the F1 FPGA, is to *selectively map AXI slaves to AXI masters*, thus avoiding a full-scale internal all to all interconnect. This helps the AXI interconnect become simpler, and thus economize on resources. What one has to do is to map to each master only the peripherals that actually need to be accessed, based on the UNILOGIC architecture's signification. For example an accelerator need not have access to the accelerator controllers, as it can only request for memory accesses, and need not encompass the ability to configure

TABLE 4.5: Network FPGA (F1) resources consumed for various AXI interconnect approaches

Strategy	**Total Static**	**Central AXI**
2xQFDB topology, 4 slots	22.03%	11.30%
full topology, 2 slots	25.52%	12.55%
full topology, **3 slots**	29.32%	15.85%
full topology, **4 slots**	34.15%	19.07%
full topology, 4 slots, **4 PS-ports**	44.05%	27.85%
full topology, 4 slots, 4 PS-ports, **2-Level AXI** (1xL1 + 3xL2)	21.92%	6.41%
full topology, 4 slots, 4 PS-ports, 2-Level AXI (1xL1 + **2xL2**)	27.62%	11.50%
topology, **3 slots**, 4 PS-ports, 2-Level AXI (1xL1 + 2xL2)	**23.13**%	8.45%

peripherals.

Under this optimization, and along with all the suited ones from those discussed before, we can see the resulting resource overheads in Table 4.5. For a simple 2-QFDB topology, i.e. only one added network interface that leaves the F1, as we see in the first row we get to spent 22% and 11.3% of resources for total static and AXI interconnect respectively. If we move on with the complete topology, incorporating all required network interfaces, we reach up to a total of 34% while still supporting four accelerator slots, as seen in the 4th row. However we we can economize on resources if we opt for less accelerator slots in this FPGA, which reduces otal overhead to 29% for three slots (3rd row), and about 25% for two slots (2nd row). If we opt to support four accelerator slots and in the same time encompass four parallel PS ports to access memory, we end up with an inordinate 44% of logic overhead for the static part (5th row), owed mostly to the pricey AXI interconnect, now greatly enlarged and consuming 28% of resources.

Deploying the hierarchical AXI interconnect approach described before, we can greatly economize on resources, as seen in the following rows of Table 4.5. As the targeted AXI interconnect becomes larger, which is the case for F1, this technique becomes even more effective. In the sixth row we can see how static resource overhead significantly drops when 2-Level AXI is incorporated, with this design incorporating in parallel all the configuration optimizations described in detail above. The total static resources now constitute just about 22%, while the AXI itself accounts for about 6.4%.

It is important to mention here that as noted in this row in parentheses, this is achieved with 3 blocks at the L2 AXI. Actually, in such more enlarged and complicated interconnects, it is justified to further investigate on *the level or parallelism offered in the Level-2 AXI interconnect*, leading to available alternatives that can evenly be supported in an effective manner. The selection of the alternatives to be implemented can be carried out depending on the usual scenarios expected to be operated on the platform. Looking back in Figure 4.36, the 6th row of Table 4.5 pertains to a case that excludes the 4 memory-related paths in the middle. Instead, only one such path exit the L1 AXI, and it then enters an additional third L2 block, that would span it into the four required paths. However this could prove efficiently useful only in the cases that the accesses through this single path are not expected to concurrently require high memory throughput.

As this is not the case for our specific accelerator scenarios that seek for the highest memory performance available, we move on to another approach, reported in the next (7th) row. This approach again allows for the four parallel paths to memory. Static overhead now reaches 27.6% and 11.5% for total logic and AXI interconnect respectively. This constitutes a reasonable implementation alternative, which we deployed in many of our tests. However, our architecture needs to support reconfiguration slots, and in such a case uniformity of the slots becomes of high importance. We thus opted for a less congested static design, which includes three accelerator slots along with three accelerator controllers. These accelerator slots can be of the exact same size as the slots in the other three FPGAs. This reduced inclusion of modules results in a design that requires about 23% for the total static logic, with the hierarchical AXI interconnect contributing 8.45%.

4.2.6 Introducing AXI Address Mapping to the multi-gigabit Link Interfaces: The Chip-to-chip module

Looking back to Figure 4.17, on the right side we can see the chip-to-chip (C2C) IP logic blocks, depicted in yellow. Such a C2C block is separately depicted in Figure 4.37. It incorporates a multi-gigabit transceiver, offered along with the FPGA fabric. Through this transceiver, a serial link to the outside world, i.e. crossing the FPGA boundaries, is offered, so we can also consider these blocks

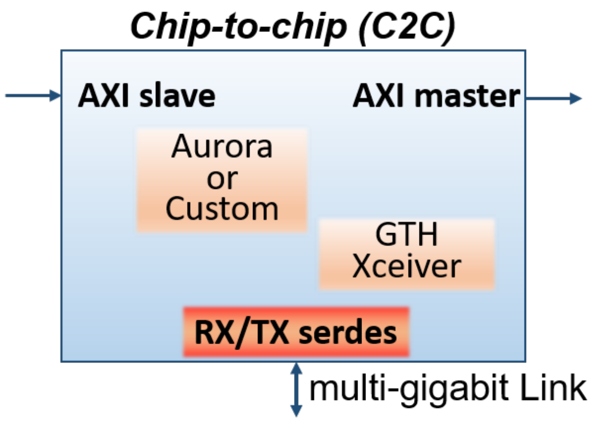


FIGURE 4.37: The Chip-to-chip (C2C) logic block, enabling uniform access to remote FPGAs, by offering an AXI address mapped interface to the mutli-gigabit transceivers

as the interfaces to the network. Briefly elaborating on the transceivers, FPGA vendors introduce custom, hardware interconnection primitives, that can achieve highly efficient, serial communication. Proper serialization/deserialization (serdes) is performed. The Xilinx devices we currently deploy, include the so called GTH transceivers, able to reach up to 16.3 Gbps, while being able to achieve low latency. The C2C is responsible for adapting the intra-FPGA AXI interconnection protocol into the protocol used by the transceivers, in order to offer off-chip communication. In other words it converts a parallel, address-mapped AXI transaction, to a serial one that matches the transceiver's serial communication protocol and vice-versa.

Through the C2C module, the serial links (GTH transceivers) of an FPGA, get presented to the rest of the FPGA components, as a usual address mapped peripheral. So a processor, as well as any FPGA component, e.g. the hardware accelerators, can initiate read/write transactions that address C2C modules. Thus, through these modules, they can indirectly access a remote Worker node, i.e. a remote FPGA, efficiently and in a transparent manner, passing through the serial GTH transceivers. Hence, components of each Worker are able to communicate seamlessly, no matter if logic resides on the same or discrete FPGA device. They can indeed operate just as if they constituted a single, vast Worker.

Multiple C2C blocks within a single FPGA can have distinct AXI address mappings. As each C2C orchestrates a dedicated GTH transceiver, and a transceiver drives a serial link, each such link can then be connected to different remote FPGAs. This way we have the ability to realize a topology of Workers. As mentioned above, in the example of Figure 4.17, only four C2C blocks are depicted for simplicity. Such a number of network interfaces, could correspond to a miniature topology. For example, it could designate three of the corresponding connections to each of the QFDB's three remaining FPGAs, and a fourth one to a remote board. So it would just support a double-QFDB interconnection, i.e. a double-QFDB prototype. More C2C blocks should be deployed to support more complex topologies of QFDBs. This is realized in our actual prototype, however not displayed here for clarity. Each FPGA device of the ones we deploy includes 16 GTH transceivers, and the Network FPGA is the one that interfaces the QFDB to the rest of the world. We can thus deploy all of the transceivers at this point, in order to achieve favorable topologies.

4.2.6.1 Details on the Exploration for Effectively Encompassing Transceivers Through the C2C Modules

The Aurora IP updates and wrapper code enhancements

The C2C block fully supports the latest AXI4 protocol. It supports bursts, and multiple pending requests, while it supports configurable Address and Data widths, as well as configurable number AXI ID and user defined bits to be send along with the data. Internally, it interfaces to the transceivers by either encompassing the Aurora protocol, or using a novel, very low-latency, custom transceiver communication module which enhances efficiency as discussed in the optimization related section 5.2. The initial C2C we deployed, uses the Aurora protocol, which is deployed in the design through an IP provided by Xilinx. However, as the Aurora based C2C was much earlier built, for designs supporting the UNIMEM architecture, an update of the Aurora IP was needed in order to be used under later tool versions. As the updated Aurora protocol included a partially diverse interface, changes as well as testing and verification cycles were needed. During this process, and as the distributed accelerator scheme of the UNILOGIC architecture puts a lot of strain to the interconnection points, some existing hidden bugs having to do with the custom design surrounding the Aurora were exposed and resolved.

Offering Support for Various C2C Bandwidths

Furthermore, the QFDB links, as mentioned, are able to support speeds of up to 16.3 Gbps on board, i.e. among FPGAs of the same QFDB, reaching the maximum rate supported. This has to do with the very precise and effective design of the

QFDB. On the outgoing links, passing through SFP+ cages and corresponding SFP passive copper cables, a 10 Gbps maximum rate is imposed due to the cabling specification (or to be more exact, reaching up to a maximum of 10.3 Gbps). So the initial C2C inherited by the UNIMEM architecture, had to be lead to segregated versions. We have build C2C versions for 16.3 Gbps and 10 Gbps, as well as version for other frequencies, which proved greatly useful during the exploration process. Each such version required different, custom setting for the Aurora protocol, and different ways to instantiate the GTH transceivers, as this also depends on targeted speed. It also includes extensive examination and targeting debugging to fully verify. In parallel, many special Bit Error Rate Testing (BERT) was comprised, deploying Xilinx tools, and custom configurations depending on the platform and intercommunication scenarios to be tested.

Transceiver Clocking Support for Multiple Transceivers of Diverse Configuration

Another troublesome complication at this point, has to do with transceiver clocking. Instantiating a single or a couple of transceivers in an FPGA is one think, while trying to deploy many, or even further all of them becomes a much complicated task. The transceivers are sensitive circuits, and in order to be operated on maximum speeds, they require dedicated clocks and properly designed clocking infrastructure to optimally minimize clocking discrepancies such as excessive skew. In order to compose a proper clocking scheme, certain constraints are imposed by the design tools. In brief, the transceivers of a ZYNQ device are grouped to quads. Then these quads are separated to a left and right side. Each side includes dedicated but limited clocking resources. These primitives should be properly encompassed if many transceivers need to be instantiates and thus share the scarce resource. And the configuration becomes further perplexed as GTH transceivers in the same quad may need to have discrete operating speeds. This required a lot of additional changes to the original C2C block: a) the GTH designated clocks entering the FPGA, had to be processed by proper GTH-related clock buffers that create a single clock from a differential one, b) the clocking primitives, as well as related logic (as the GTH_COMMON primitive) had to be excluded from the C2C block, involving various other alterations, c) a common clocking primitive had to be instantiated per left/right row of transceiver quads, d) a single C2C per row had to be configured as clock "master" and the rest as clock "slaves". Again a lot of low level testing was needed, as well as monitoring through the chipscope tool flow, and finally validation through stressful testing by incorporating remote acceleration scenarios.

Replacing the Aurora IP with a Custom Low-latency Module

A very important characteristic of the C2C module is that of the introduced latency. Standing in the boundaries of each FPGA, a considerable amount of latency is added in order to properly Serialize/Deserialize parallel traffic into the multi-gigabit links driven by the GTH transceivers. A series of measurements and evaluation, identified the Xilinx's Aurora IP, embodied within the C2C, as the component that contributes most of the measured latency. One reason is that a lot of extra functionality is included in the Aurora IP, in order to be adaptable in various possible configurations, however not applicable in our case. Nonetheless, there exists no other simpler, out of the shelf module offered, in order to make proper use of the available transceivers. This "Aurora-aided" version of the C2C was measured to introduce a latency of about $220ns$, and we should bare in mind that this latency is appended at every chip (i.e. FPGA) boundary. This means that it gets added four times in a cross-chip round trip time (RTT). We moved on to replacing the Aurora IP with a custom, low-latency Serialization-Deserialization (SerDes) IP block, that directly wraps the GTH transceivers and incorporates the respecive low level protocol. This was designed by the CARV team at FORTH, and after initial validation through simulation, it was deployed in the UNILOGIC infrastructure for actual hardware testing. Initial low level bare-metal testing revealed some first problems, and after improvements the new C2C was stressed under the much more demanding real accelerator traffic. Corner case bugs where then revealed and proper modifications were applied. Also, the chipscope tool flow used to enhance verification, was also employed to accurately measure the latency overhead for the updated C2C. This "custom-serDes" C2C was measured to offer substantially reduced latency, now contributing a mere $90ns$ at each FPGA boundary. As mentioned this latency is quadrupled in a Round-Trip measurement, so it becomes of most importance under the UNILOGIC architecture, encompassing remote accelerator invocation, and symmetrically remote data access by accelerators. As these paths get greatly affected by FPGA crossing latency, this can easily ruin any effort for a meaningful remote acceleration, and likewise effective resource unification. More details on the aggregate latency and how this affects the UNILOGIC infrastructure are given in section 5.1, regarding optimization.

Further Possible Enhancements Through Link Bonding and New Flavors of Transceivers

As data transfer, or equally throughput availability, is a common cause of bottlenecks, additional research that targets optimizations for inter-FPGA communication throughput is beneficial. A favorable approach is to further enhance the C2C IP block by incorporating link bonding. As we have seen, the FPGAs offer many transceivers that can in turn drive serial links, while pairs of such links are deployed per connection in our prototype implementation. Having more than one link per connection provides flexibility, as they can be used to support different topologies, separation of communication protocols, etc. One other advantageous usage scenario of multiple links, is that of throughput aggregation. An effective way to do so is through link bonding. What this means is to have a pair of GTH transceivers (or more) deployed in parallel, and presented as a transceiver with double throughput. This is more difficult to implement than just using the two transceivers/links separately, as efficient off-the-shelf solution are not available, and thus customization analysis need to take place. A pair of bonded links interface through a single AXI interface to the rest of the FPGA logic, which appears as if it encompasses a single transceiver of double the throughput. This allows transparent utilization of the bonded links, and, in case of link pairs, doubling of link capacity. This now provides $2 \times 16.3Gbps = 32.6Gbps$ of throughput for intra-QFDB communication, while for inter-QFDB communication, it translates to $2 \times 10.3125Gbps = 20.625Gbps$.

A link bonding scheme using the Aurora protocol, and customizing through the Xilinx Aurora IP was employed and verified. A drawback of the Aurora IP worth mentioning is that it renders transceivers under link-bonding as one way communication paths, i.e. either as senders or receivers. This may be useful in some cases, however not advantageous in a broader aspect, so as to perfectly fit our UNILOGIC approach. More interestingly, effort was invested on custom serDes versions of the C2C module, i.e. deploying a custom transceiver wrapper as before, which additionally bonds a pair of links. This entailed a lot of added techniques concerning e.g. synchronization and clocking, with great effort from the CARV laboratory at FORTH. This module, after initial cycles of simulation and adjustments, was employed and tested in real hardware, and also incorporated in the UNILOGIC prototype's infrastructure and has reached a functioning state. This revealed that custom link bonding through GTH transceiver pairing constitute a feasible approach, even if this was not deployed and fully evaluated on the full platform, as a final stage of verification is still to be taken care of. Nevertheless, this causes no drawbacks, as the prototype implementation is either way performing gracefully.

Resource Unification Potential, Based on the Latest State-of-the-art Transceiver Availability

One final important aspect we examined and would like to cover in brief, is that of FPGA transceiver availability. As mentioned, the ZYNQ MPSoC devices of our platform, being the state-of-the-art at the time of building the prototype, embody the GTH transceivers, which can reach a maximum of 16.3 Gbps. Furthermore, these devices constitute a viable option concerning procurement cost. In parallel, there is also a continuous blooming of enhancements on this field. FPGA vendors relentlessly try to push the bandwidth of the hardware interconnection primitives to the maximum. They do so, as the offered connectivity is one of the main advantages of FPGAs, as it allows fast cross-chip interfacing, with limited protocol delay overheads. As a first example, few of the current Xilinx's MPSoC devices encompass the upgraded GTY transceivers, which can offer up to 32.75Gbps of throughput per link. PCB boards in turn can deploy Quad-SFP (QSFP) or Quad-CFP (CFP4) connectors and respective cables, which allow 4 such transceivers to be driven through a single connector and correspondingly through a single cable, usually able to reach up to 100 Gbps per single-cable. The latest GTM transceiver flavor, included in the Xilinx's Versal family of SoCs [49] operate at data rates up to 58 Gbps. Importantly, appropriate hardware as the previously mentioned custom link bonding would likewise prove beneficial to properly utilize such enhanced interconnectivity. The Altera FPGA vendor (acquired by Intel) on the other hand offers the GX/GXT/GXE transceivers, similarly reaching up to 57.8 Gbps. It also delivers the 10nm Agilex FPGA SoC family, which features transceivers capable of up to 112 Gbps [19]. All this continuous improvement on the FPGA interconnection hardware primitives is a great match for our UNILOGIC architecture, as it is quite promising to even further elevate transparent FPGA resource unification.

4.2.7 Summarizing the UNILOGIC-incorporating FPGA design

As a summary, the presented UNILOGIC architecture allows for the implementation of a unified and virtualized multi-FPGA platform, where all the memory and the reconfigurable accelerator modules appear uniformly in a global address space. Any node that seeks access to any memory or accelerator in this address space, issues the same commands as if everything was local, i.e. as if the complete system comprises of a huge contiguous reconfigurable fabric along with the corresponding enormous memory resources. The hardware primitives implemented are responsible to transparently forward any transaction, thus deploying distant accelerators as well as remote memories without any user application intervention. All the interconnection infrastructure is designed with efficiency in mind, both in terms of latency as well as throughput, allowing any intersecting module to simultaneously perform effectively. This leads to parallelizing of all the resources, enabling accelerators to run in parallel, in local as well as remote FPGAs, in a multi-CPU parallel system fashion. In order to verify the correctness and efficiency of our approach, and evaluate performance both in terms of execution time as well as power consumption, we have performed several tests at various hardware and software levels (e.g. from bare-metal to OS running application, and from single FPGA to multi-FPGA scenarios), as our prototype evolved. These tests, for example, include intra- and inter-node memory transfers, configuration and parallel invocation of local and remote accelerators, as well as accelerators and processors using both local and distant memories. Our testing and evaluation process is presented in Section 6

4.2.8 FPGA Utilization

Keeping our design's demand on resources as low as possible becomes of prominent importance in the UNILOGIC context, as it allows for more resources available for the accelerators. This overhead of resources spent corresponds to the "static" part of the design, i.e. the part that will not be reconfigured at run time, as can happen with the accelerator slots. It pertains to all the FPGA's hardware design described previously, i.e. the interconnection scheme and any accelerator surrounding logic, excluding only the reconfiguration slots. For the complete and final prototype design, which incorporates the full interconnection scheme and any presented optimization, the static design overhead can be seen in Table 4.6. It is kept to a mere 23% of LUTs for the Network' FPGA, i.e. the most congested one, including the full topology realization, and as low as 12% of LUTs for the remaining three QFDB FPGAs. The 11% difference between these two percentages, mainly has to do with the quite larger AXI interconnect deployed in the Network FPGA, and to a smaller extent with the higher number of C2C modules. In absolute numbers, a Network-FPGA's overhead requires 63K LUTs (23%), 72K CLB Registers (13%) and 105 BRAM tiles (11%). The other three FPGAs' overhead falls down to 33K LUTs (12%), 33K CLB Registers (6%) and 20 BRAM tiles (2%) It is also worth mentioning that absolutely no DSPs are spent in this "static" part. This is important, as DSPs are in most cases the critical resource for implementing hardware accelerators.

TABLE 4.6: Static overhead per FPGA: the resources consumed for the static portion of the UNILOGIC implementation

FPGA	LUTs	CLB Registers	BRAM tiles	DSPs
F1 (Network)	23%	13%	11%	0%
F2, F3 & F4	12%	6%	2%	0%

4.3 Runtime Partial Reconfiguration Support & System Software Flow

In order to introduce runtime partial reconfiguration in our architecture, we had to deploy proper hardware modules within our design, and in parallel develop the proper software that efficiently utilizes the additional hardware functionality. The architecture itself was thoroughly devised, with reconfiguration inherently envisioned, so that the needed extra hardware can easily be introduced. As for software, it aims to both offer the low level device drivers, as well as the high level distribution of tasks among available resources. We describe the changes in the Operating System and the Runtime support developed on top of the hardware infrastructure so as to facilitate the programming of the parallel heterogeneous platform, and also briefly present the way this software supports and automates the partial reconfiguration process.

The accelerator slot (Partial Reconfiguration slot, or PR-Slot) interface specifications was part of this thesis, as well as the PR-slot isolation hardware described in section 4.3.1. Also a lot of co-designs efforts took place, pertaining to PR-slot deployment and partial bitstream propagation. The PR-slots themselves, as well as the partial bitstreams and partial bitstream manipulation was contributed by the University of Manchester. I also contributed in the development of the software stack, which otherwise was mostly contributed by TSI Institute in Chania, Greece. In addition, all the related dynamic partial reconfiguration testing, as well as all the scenarios executed and reported herein were performed in a close collaboration between the author of this thesis and the TSI team in Chania.

4.3.1 Hardware Infrastructure for Remote Partial Reconfiguration

One of the main capabilities that need to be added to the architecture, in order to offer the partial reconfiguration attribute, is that of transferring a partial bitstream on to the FPGA designated slots. What is most commonly used for this reason is the Processor Configuration Access Port (PCAP). This is a primitive provided by Xilinx, and already included in the Processing System of our Zynq MPSoC. PCAP is used to configure the FPGA part, also referred to by Xilinx as Programmable Logic (PL), from within the Processing System (PS). Using this path is not only the most common mechanism for partial reconfiguration, but also comprises the primary configuration mechanism itself, for the FPGA part (PL) of the Zynq Ultrascale+ MPSoC, as in the normal configuration process of the FPGA, the Processing System actually delivers bitstreams to the PCAP. This can be seen on the left of Figure 4.38.

Directly managing remote partial reconfiguration, using ICAP

However, to manage partial reconfiguration completely within the PL, partial bitstreams can also be delivered to the Internal Configuration Access Port (ICAP). This is a Xilinx provided block that can be included in the FPGA design, as seen on the right of Figure 4.38. The ICAP is essentially an internal version of the SelectMAP interface [48, 86]. What is most important for our architecture, is that by allowing partial bitstreams to flow through the reconfigurable logic, without intervention by the processing system, we can also perform this process remotely.

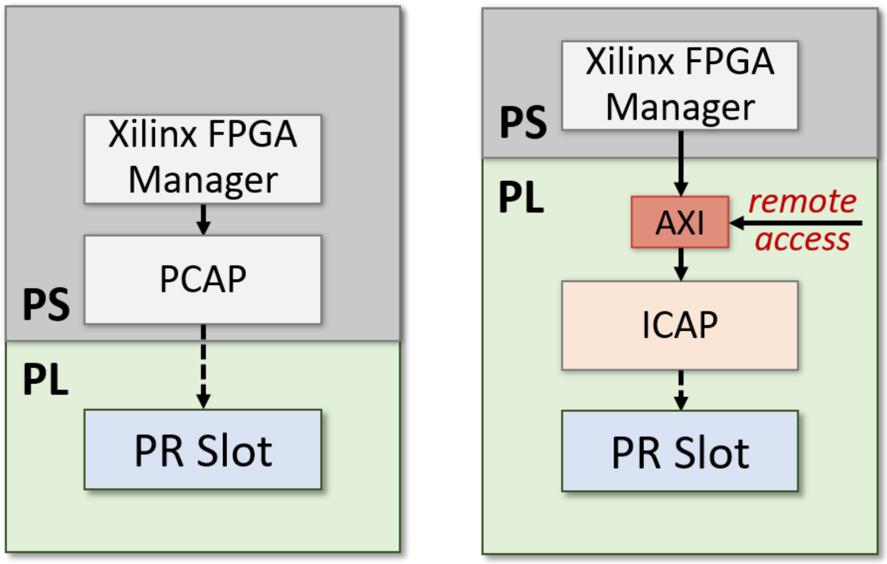


FIGURE 4.38: The PCAP and the ICAP controllers constitute the two alternatives for FPGA partial reconfiguration

That is we can send a partial bitstream, in the same way we send data, over the UNILOGIC architecture, and thus target any available reconfigurable slot in the system.

This can be seen in Figure 4.38. The PCAP resides in the PS part of the MPSoC, and so partial reconfiguration must be initiated within the local PS, while the ICAP resides in the PL part and includes an AXI interface. Having an AXI interface, it can be accessed by either the local PS, or by any other entity in the whole system. By initiating suitable AXI transactions, either from the local or from a remote Node, and aided by the UNILOGIC architecture, they can be routed in order to drive the specific ICAP block.

It is useful to mention that the PCAP and ICAP interfaces are mutually exclusive and cannot be used simultaneously. Switching between ICAP and PCAP is possible, but we must ensure that no commands or data are being transmitted or received before changing interfaces. What we usually do in our implementation, is to allow the usual PCAP configuration process for the initial FPGA configuration, and then as the configurable/programmable slots (PR Slots) are available in the PL, we switch the operating interface by disabling the PCAP and enabling the ICAP one.

Having added the ICAP in our FPGA block diagram, actually just adds one more local peripheral, and a corresponding configuration address space of 64 KB.

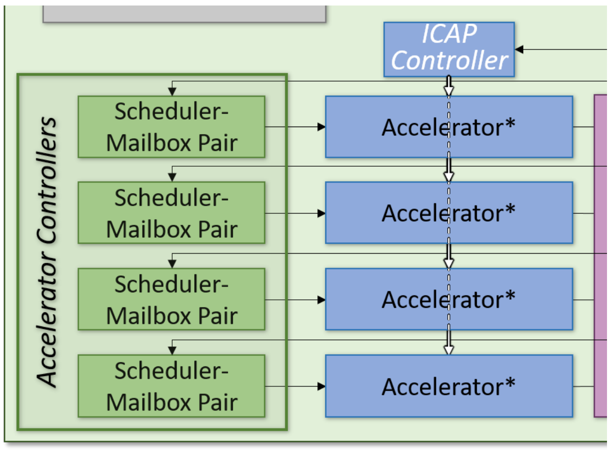


FIGURE 4.39: The FPGA block diagram detail, with the ICAP controller included in the reconfigurable logic, in order to partially reconfigure accelerator slots

The partial bitstream is passed to the ICAP, and then the proper accelerator slot will be programmed, as seen in Figure 4.39 which is another augmented detail of the original FPGA block diagram.

The need to temporarily disable AXI interfaces using decoupling

During the partial reconfiguration of the FPGA, used to load a new hardware accelerator, all signals crossing the boundary between the partial and the static regions must have predefined values in order to both keep the static FPGA logic function trouble free, even if the reconfiguration generates unpredictable values to its outputs, and also prevent the hardware accelerator to enter an unknown state when, during the reconfiguration process, it receives unpredictable values from the static portion.

A decoupler block is used to prevent such problems. This is mainly build by two multiplexers as shown in Figure 4.40. A 'Decouple' signal controls the inputs/outputs to/from the hardware accelerator, and can choose between the actual signals or a predefined value set to pass through the interface.

The decoupler shown in Figure 4.40 must be used to control each signal of the interface generated by Vivado HLS for the hardware accelerator. This contains one AXI-lite bus to control the kernel and one AXI bus to communicate with the system memory. For this reason, this IP can be parameterizable to decouple all the signals in the interface, both in terms of interface widths and for the predefined values to be used during the partial reconfiguration process. The decoupler can be address mapped, and then again the decoupling can be controlled by commands

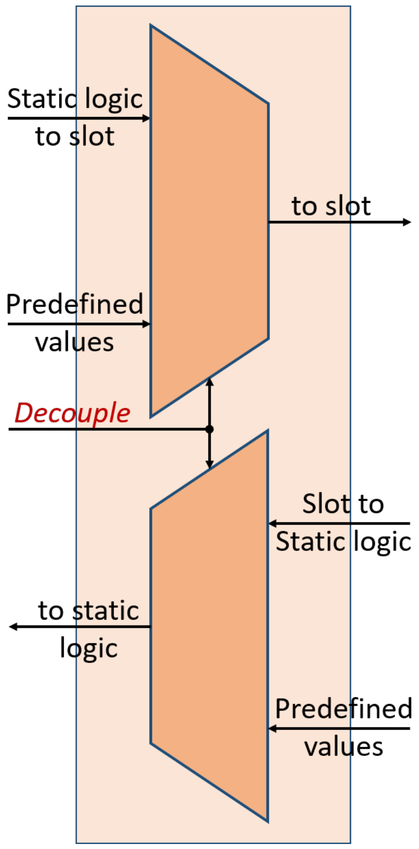


FIGURE 4.40: The decoupler block intervenes in order to protect the FPGA logic from faulty signals produced during the partial reconfiguration process

that write to the proper addresses.

Accelerator slot isolation with AXI-registers for improved timing

The PR slots themselves are devised by the University of Manchester, and details can be found in [91, 89, 40, 113]. However due to internal PR Slot architectural reasons, a problem arose as to the timing closure. What we understood after though redesign and testing, was that the tool, when compiling the static logic was oblivious of the –still empty– PR slot logic. And the same applied for the process of constructing a partial bitstream that would fill a PR slot. In order to bypass these issues, and to achieve efficient timing even with PR slots, we added registers at both sides of the PR slot. This can be seen in Figure 4.41, where the decoupler is also shown, embracing the PR slot related logic. Of course just placing registers in an AXI interface path, would uniformly delay all the control signals, which would violate the AXI protocol, adhering e.g. to specific constraints as for the timing between requests and associated responses. Instead an "AXI register slice" IP is deployed on both sides, which both isolates the PR slot from the static

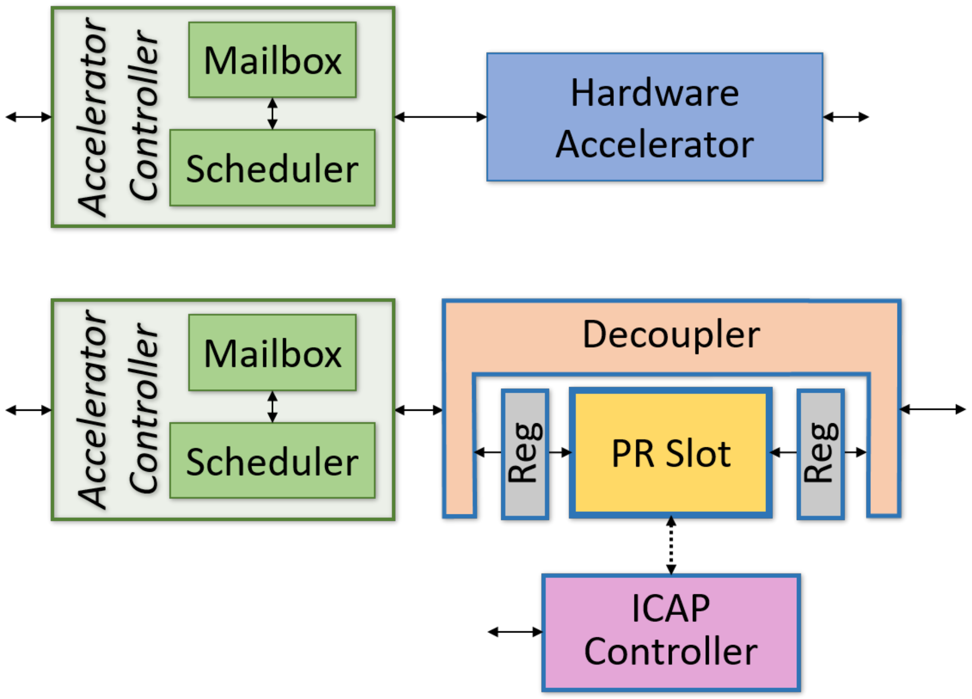


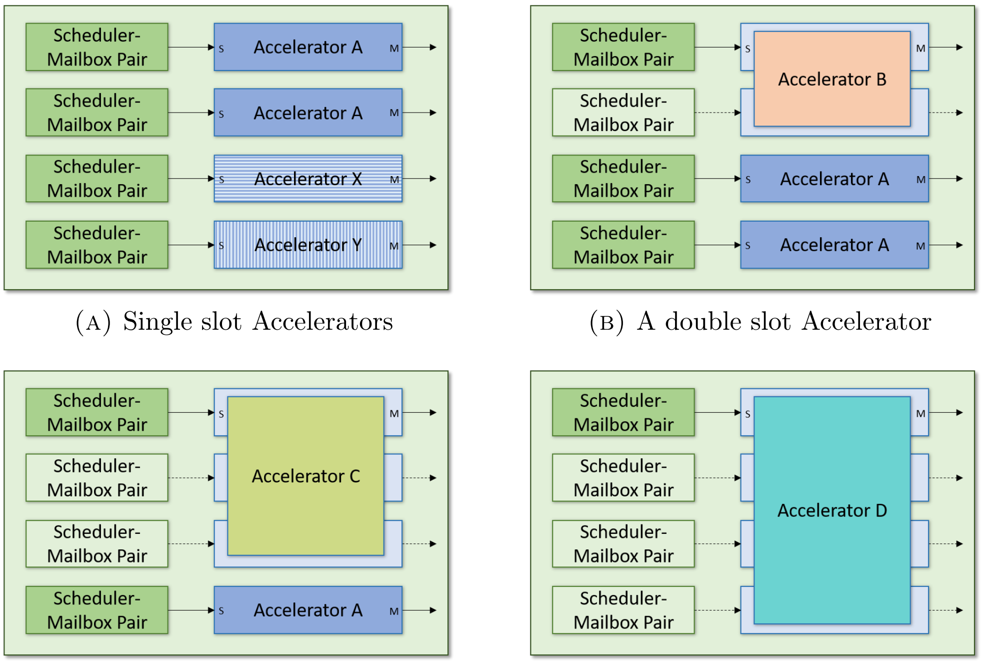
FIGURE 4.41: The accelerator is exchanged for a Programmable Accelerator Slot (PR Slot) to allow partial reconfiguration. A decoupler, along with AXI register slices, have to intervene in order to have an efficient solution

logic with registers, and properly preserves the AXI protocol attributes.

Offering the feature of accelerator slot merging

As a last thing, we would like to refer to the briefly aforementioned slot merging capability offered. An important aspect when offering resources for a purpose, as is the case with accelerator logic, is to efficiently share available resources among possible occupants. There are cases when many small accelerators, either different or all the same, need to use parts of the logic in order to execute task. This is why we have selected to upscale our implementation by deploying four independent accelerator controllers, that run in parallel. They all include dedicated translation blocks, attach to a distinct Processing System port to access memory, and each can be disjointly configured to support disparate accelerators on their controlling slot. Such a case is depicted in Figure 4.42a, with four single-slot sized accelerators occupying the slots. They can be either of the same accelerator core, or of different ones.

Nonetheless, one equally popular scenario, is that of having to host larger accelerators cores, that require a higher amount of logic resources. Just having



(A) Single slot Accelerators (B) A double slot Accelerator

(C) A three slot Accelerator (D) A four slot Accelerator

FIGURE 4.42: Accelerator slot merging, allowing diverse accelerators, even with resource requirements greater than a single slot, to jointly occupy a varying number of available slots

the available logic spread in disjoint accelerator slots, would decisively disallow such valuable as well as common cases. In order to support favorable utilization of the available resources, for either scenarios, the static logic supports slot merging. As seen in Figure 4.42, more than one consecutive slots can jointly offer their resources, in order to host a single accelerator of proportionate size. When slots are jointly deployed, only one accelerator controller, i.e. scheduler-mailbox pair, takes charge of the combined slot. All other controllers corresponding to he merged slots get disabled. For example in 4.42b a double-slot sized accelerator occupies the two first slots, and only the first of the two accelerator controllers remains active. Certainly all the other controllers can remain active in order to manage corresponding slots. In 4.42c and 4.42d we also depict cases for larger accelerators of increasing size, occupying three and all four slots respectively. We should also mention here for completeness, that proper care has also been given on the other side, that of the partial bitstream, in order to conduct valid partial bitstreams for multiple slots, however not in the scope of this thesis.

Accelerator slot defragmentation

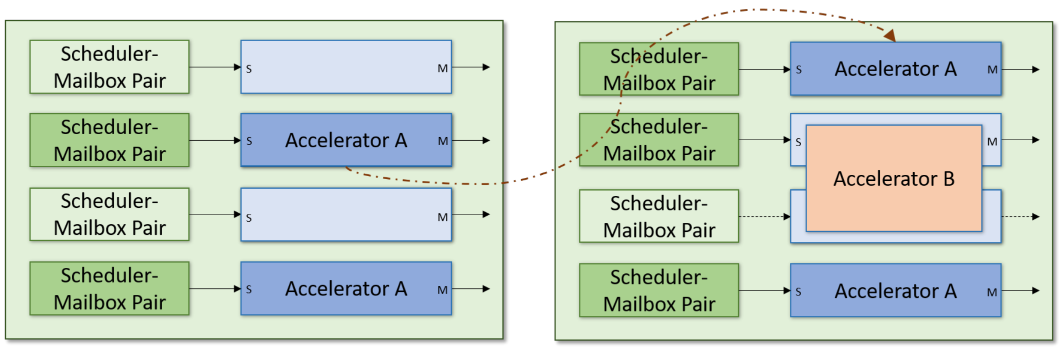


FIGURE 4.43: Slot migration in order to overcome fragmentation

All this accelerator slot resource handling, reminds that of memory management techniques. So, as it occurs with memory allocation, likewise for accelerator slot allocation, resource fragmentation can occur. For example, in Figure 4.43, we see a snapshot in time, at which we have come to the situation where, as seen on the left, two small accelerators occupy alternating single slots, with the other two being unused at the moment. If a larger, double-slot accelerator needs to be hosted, although the volume of resources exist, they are fragmented. In this case, migration of accelerators can be used to resolve this issue. For this example, one accelerator can get relocated to another slot, leaving two adjacent slots that can now be jointly used. In the right of the figure, the resulting configuration serves all accelerators concurrently, utilizing all slots, and with just the needed accelerator controllers activated.

4.3.2 Software Support for Runtime Partial Reconfiguration

UNILOGIC can support numerous hardware accelerators implemented in different Compute Nodes/Workers and reconfigured at execution time. In that respect, we have implemented in our prototype, higher-level (software) mechanisms needed to allow for this reconfiguration to be efficiently supported. This includes the mechanisms so as to transfer the partial bitstreams into the corresponding FPGA slot(s). In addition, in order to increase the efficiency of our approach we have to optimally assign/manage the reconfiguration resources/slots at real-time. Finally, it is important that the system is user-friendly and that most of the work takes place under-the-hood, in other words, a perspective user should not be concerned with how a new accelerator is going to be introduced to the system. Similarly, the system must be able to scan through a number of different accelerator modules and select the one(s) that serve the executed application best.

The reconfiguration-related tasks have been implemented within the adopted prototype as part of the custom Linux OS hosted on the PS of the Network FPGA of a QFDB as shown in Figure 4.44. The OS is crucial since it introduces a layer that corresponds to the application activity, i.e. *User-Space*, while isolating, therefore protecting critical system aspects such as its physical domain. In such a way the user introduces an application, parts of which will be executed on hardware accelerator(s) without being burdened by details on how to perform the actual reconfiguration slot programming and/or, for example, running the risk of directing bitstreams towards forbidden areas of the FPGA programmable logic.

The OS is complemented by an appropriate (Character Device) Driver, Figure 4.44, which is the bridge between what the user desires to do and how this is executed on the the actual hardware. The device driver is located in the kernel space of the operating system, which is the in-between layer among user-space and the physical domain, and uses well known APIs, such as *copy_ to_ user()* for user-space and kernel space communication. On the other hand, the communication between the kernel space and the physical domain is done via the use of virtual addresses, i.e. data from user-space are passed onto virtual addresses, therefore, the risk of accessing the wrong physical address is eliminated.

The reconfiguration of the resources is done through the ICAP module provided by Xilinx, which is responsible for allocating the partial bitstream at the appropriate coordinates in the reconfigurable fabric. Therefore, the Character Device Driver (CDD) we have designed communicates and manages that particular module in order to implement the partial reconfiguration feature of our system. Additionally, it is either the user or an automated runtime system process that selects the appropriate partial bitstream as well as the accelerator slot, in the parallel system, that the bitstream will occupy.

Hence, these two pieces of information are propagated onto the kernel space. Subsequently, the ICAP device driver is registered with the Linux kernel and a virtual address is assigned to the ICAP hardware module. That particular address is then used for writing the partial bitstream into the ICAP's FIFO buffer and after this is completed, the partial bitstream is programmed into the reprogrammable fabric of the target FPGA. When this is done, the user or the automated process

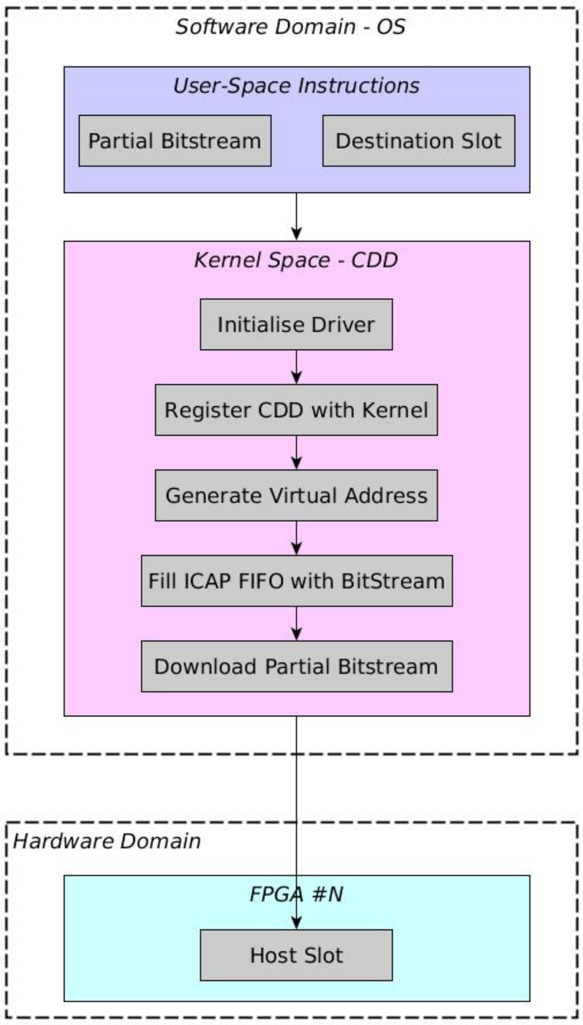


FIGURE 4.44: Example Dynamic Partial Reconfiguration Flow

is notified that reconfiguration has been completed and application execution can commence.

If the user does not want to get involved with where the hardware accelerators will be implemented, the designed automated runtime process assumes responsibility for this task, Figure 4.45. The designed runtime daemon uses two types of information: One has to do with the user application(s) while the second has to do with the status of the system's hardware resources.

Regarding the information from the user application, the runtime analyses the application and identifies which algorithmic elements are suitable for hardware implementation. Such elements mainly have to do with computation tasks rather than control ones; in other words hardware acceleration is selected for the computationally intensive tasks. Next, the system's *Acceleration Library* is consulted

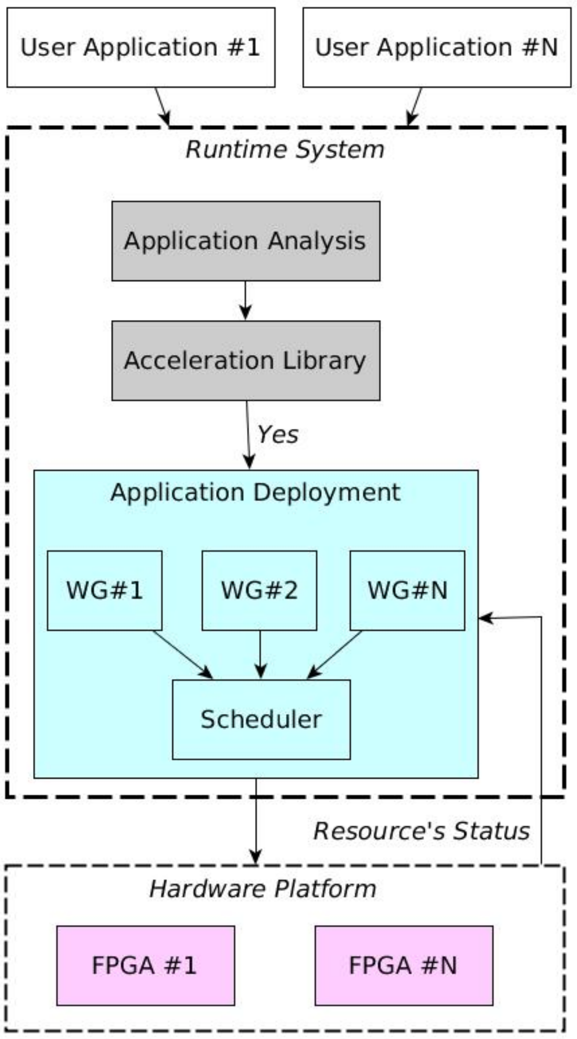


FIGURE 4.45: High-level view of system operation

in order to identify the partial bitstream that correspond to the task/sub-task selected for hardware implementation. In the event that the acceleration library is missing a suitable partial bitstream for the desired algorithm, the user has the ability to refer to another tool [38] which allows for the generation of new partial bitstreams that can be used as accelerator modules and, consequently, become part of the system's acceleration library.

Subsequently, the runtime system takes into account the *Resources' Status* in order to partially reconfigure the appropriate FPGA slots according to the process presented in Figure 4.44. Since in our prototype we have used applications developed in Open-CL, the developed runtime daemon operates in terms of *WorkGroups (WGs)* which are assigned onto different FPGA slots via the *Scheduler*. The runtime uses, the sizes of the bitstreams, as well as the historical data from previous executions so as to perform this assignment in a (near) optimal manner in terms of execution time and/or energy consumption. As also stated in the Introduction section, the details of the implemented hardware reconfiguration mechanisms are beyond the scope of this thesis, and the related information can be found in [91].

4.4 A Compound Example: Summing Up the Effectiveness of the UNILOGIC Implementation

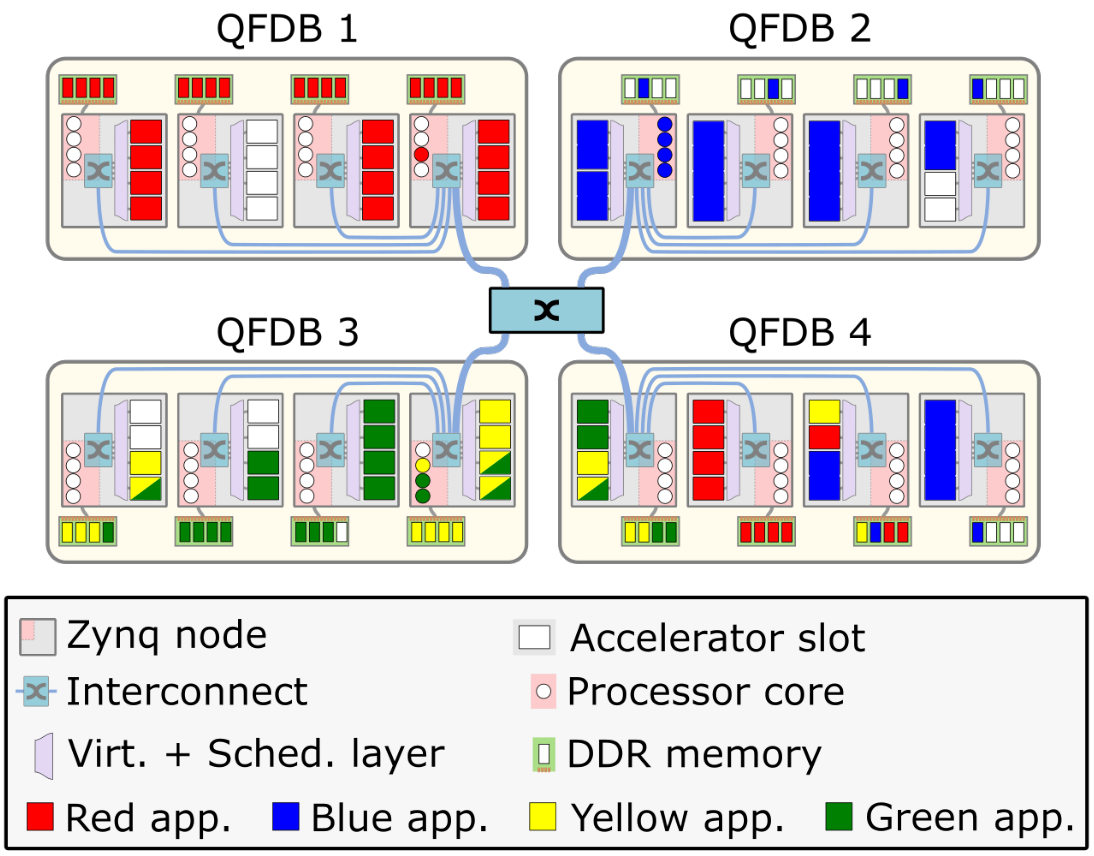


FIGURE 4.46: A snapshot of a UNILOGIC operational scenario

At this point, and just before we move on with elaborating on the systematic performance optimization process in section 5 and the thorough system evaluation in section 6, we consider beneficial to present a small recap. A brief summary of the UNILOGIC architecture's high level contributions, along with a demonstrative example including a blend of possible accelerator scenarios that the UNILOGIC architecture and current implementation is able to support.

In order to implement efficient sharing, the UNILOGIC architecture supports partitioned global address space so that a) the hardware accelerators on all the FPGAs of the system, residing in QFDB boards in this current implementation, can be accessed directly by any processor in the system, and b) the hardware accelerators can access any memory in the system. In this way, the architecture offers a unified environment where all the system resources can be seamlessly accessed by software running on any processor.

Instead of just deploying FPGAs as nodes in the system, each node is an entire sub-system including a processing unit, memory, storage and reconfigurable resources that can be accessed by any other node in the system. The reconfigurable resources are split into a static partition, which provides the communication infrastructure, and four fixed-size slots that can be reconfigured and accessed independently or combined together (slot merging), in order to support fine or coarse grain partitioning and utilization of the reconfigurable resources.

Partial runtime reconfiguration is supported, in order to dynamically reconfigure the accelerator slots. These slots can be remotely reconfigured and accessed directly by any processor in the system using the Xilinx Internal Configuration Access Port (ICAP) that resides in the UNILOGIC global address space. The tight coupling of the resources on the QFDB, along with cautious implementation of the UNILOGIC principles, allows an accelerator on any reconfigurable slot to access any DDR memory in the system with minimal communication overhead. A runtime system can be deployed, to monitor the system status and manage the reconfigurable resources across the whole platform.

As a demonstrative example, we present in Figure 4.46 a small UNILOGIC system with four interconnected QFDBs. In each Zynq Ultrascale+, the UNILOGIC firmware implements four dynamically reconfigurable accelerator slots, that are accessible to applications through the accelerator controllers that offer a virtualization and scheduling layer. Four applications share resources in this example, each differentiated by a different color. Inside each MPSoC, the four accelerator slots are depicted as rectangles, while the four ARM processor cores are depicted as circles. Each application running on a processor fills it with the corresponding color, and the same happens when an accelerator slot is deployed. The smaller rectangles outside each MPSoC pertain to the DDR memory modules (DIMMs), and likewise get filled with data by each application. In the middle, a switching element is displayed, however just to declare connectivity and routing among the QFDBs. In the actual implementation, this routing takes place inside each network FPGA, standing at the "edge" of each QFDB, and thus providing a modular and scalable approach, discarding the need for additional and disparate central interconnection modules.

Getting in more details, we see that e.g. the software part of the Red application runs on a node of QFDB 1 (red circle in the rightmost FPGA), and spreads its data in the memory of the local as well as other nodes (red memory DIMMs). Accelerators for this application (red slots inside FPGAs) are spawned close to the data, however all accelerators can also access remote memory with a moderate latency. So this Red application has also acquired resources from two additional FPGAs in QFDB 4. Another application, the Blue one, requires two coarse grain accelerators, i.e. deploys two kinds of accelerators that require respectively the merging of 2 and 4 slots. Again, as the QFDB 2 resources do not suffice, it also spawns to QFDB 4 as well.

Two more applications, the Green and Yellow, run in and share QFDB 3. Since the UNILOGIC virtualization and scheduling layer allows for seamless sharing of accelerators across applications, they also reuse a common accelerator function (double-colored slots). That is, as seen in the Figure, each of them needs to deploy a distinct accelerator, but also, both of them need to use a common accelerator core. So the double colored rectangle, is a hardware accelerator that can be shared between both software applications. Likewise, as with the previous applications, these also use resources of QFDB 4. Thus, through the execution snapshot of Figure 4.46, a summarizing view of the flexibility offered by the UNILOGIC architecture gets illustrated.

Chapter 5

Performance Optimization

The initial evaluation results, that proved the feasibility of our approach, were based on a commercial board hosting a single-MPSoC and are presented in [75], while part of these can be found in Appendix D. In this part of the thesis, we report execution results for accelerators that implement the Michelsen algorithm, which is used in the calculation of the Rachford-Rice [124] equation. A variation of this equation is specifically tuned and extensively used in the field of oil-Reservoir Simulation (RS). A high level insight can be gained through Figure 5.1, where a rock matrix is mapped to a grid model[1] for which the RS simulation predicts the oil and gas flow within the rock formation. As the initial implementation of the Michelsen kernel is not optimized for FPGA execution, manual code transformations have been applied in order to reach efficient hardware implementations [51]. The early phase of our work, included in our first publications, confirmed that the UNILOGIC architecture can, indeed, be effectively implemented in today's reconfigurable MPSoCs; even a very simple implementation on a single device triggered a 30% speedup over a quad-core Intel-i5 CPU and more than 2x reduction in energy consumption.

The Michelsen accelerator core proves even more useful for the UNILOGIC implementation on the multi-FPGA prototype, as it generates a lot of data movement, thus offering an approach that stresses memory bandwidth as well as multi-gigabit link throughput. This way we can identify bottlenecks and seek for improvements that will optimize the UNILOGIC implementation. We first optimize the implementation on a single FPGA, and then we proceed to the optimization for multi-FPGA scenarios. In particular, we initially focus on the optimization of

[1]figure by the University of Calgary, http://ucalgaryreservoirsimulation.ca/study-on-matrix-itsrole-in-controlling-oil-and-water-movement/

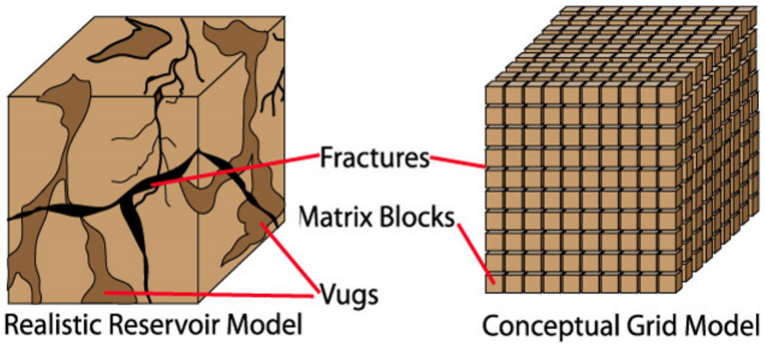


FIGURE 5.1: Reservoir simulation rock matrix and respective grid structure

a single-FPGA implementation, explaining quantitatively the performance bottlenecks and the improvements. Next, we optimize the UNILOGIC implementation in multiple FPGAs of a QFDB, optimizing the inter-FPGA efficiency so as to increase the remote acceleration performance. In this section we also explain all the steps, choices and decisions during our optimization process, through the evaluation on different implementations of the accelerated tasks as well as different communication alternatives.

I devised the testing infrastructure and related scenarios, as well as all the bare metal testing and scripting for test automation and gathering of optimization and evaluation results. I also build the complete FPGA firmware and used the chipscope tool flow to identify bottlenecks, and the investigate on the proposed solutions. This in turn led to new iterations of restructuring both accelerators and the UNILOGIC architecture in a co-design process and currying out refreshed tests. The HLS accelerators and related HLS code optimizations was the part contributed by Synelixis S.A.

5.1 Optimizations on a Single FPGA

In this case, the main challenge has been to optimize accesses to the main memory. The throughput to the memory was limited largely due to the limited throughput of the AXI-based communication between the accelerators and the DDR memory controller. Moreover, the latency to access the memory is high. We measured a latency of 300 ns (request to to first data response time), which is mostly due to the PS-to-PL interfacing and internal PS switching, as well as the processing time of the memory access by the DDR controller. The DDR memory access time itself is expected to usually contribute just about 15 to 20 ns. As verified, this remains almost unchanged regardless of the programming logic's clock speed. For example, when the accelerator was clocked at 200 MHz we measured a 60 clock cycles (cc) delay, while at 125 MHz we got 37 cc, which both give $(60cc \times \frac{1}{200}) \simeq (37cc \times \frac{1}{125}) \simeq 300ns$. It should also be noted that we have configured the internal PS clocks (CPU, buses and peripherals) at their highest supported frequencies. To work around this, we aimed to generate enough continuous data traffic to hide the 300 ns latency. A number of performance optimizations were applied to achieve this. The reported optimizations concern a continuous co-design process, involving the accelerator design, i.e. High Level Synthesis (HLS) [21], the FPGA surrounding logic, i.e. Xilinx Vivado tool, and the corresponding software application.

The initial transition from a fully-unoptimized to a first-level optimization stage is reported in [75] and constitutes the starting point in this section. At that time, commercial single-FPGA boards was used, manufactured by Trenz Electronic GmbH and deploying the exact same FPGA, yet at that time provided only as engineering samples. Subsequently, the next-level optimization has been to alleviate the impact of latency by allowing for many more ongoing data. To achieve this, we had to enhance the interface of the accelerators in order to support burst transactions. Hence, bursts of 16 accesses were introduced, with each being 256 Bytes in size, resulting in a burst size of 4 KB, i.e. reaching the maximum burst size supported by the AXI interconnect.

As a result, the execution time for a single Michelsen core, processing a sample dataset of 2400 blocks of 100K elements each, dropped from about 324 sec down to 108 sec, as shown in Table 5.1. As each element uses 16 numbers to describe an Oil Reservoir simulation grid point, these 100K elements correspond to $100K \times 16 \times 4Bytes = 6.4MBytes$ of data, and 2400 such blocks add up to $6.4\ MB \times 2400 \approx 15\ GBytes$ of data. Importantly, all reported measurements are based on this date

TABLE 5.1: Execution time and performance (sec and GFLOPS) for Michelsen cores, for various co-design optimization, on single-FPGA design versions running at 200 MHz

Time (sec)	Optimization Version	GFLOPS	step improvement
324.26	HLS optimized Michelsen	1.8	–
108.12	Burst-enabled co-design	5.4	3x
27.15	Burst & 128-bit datapaths	21.6	4x (12x)

set size, and captured on designs that are all clocked at 200 MHz.

A consecutive optimization was to increase the width of the AXI datapath on all the logic blocks and interfaces of the design. Hence, it was increased from 32 bits to 128 bits, which matches the maximum bus width supported by the processing system ports of our MPSoC device. This led to a further performance boost, with the execution time decreasing to 27 sec. The improved execution times are shown in Table 5.1, along with the equivalent GFLOPS, while the corresponding plot can be seen in Figure 5.2.

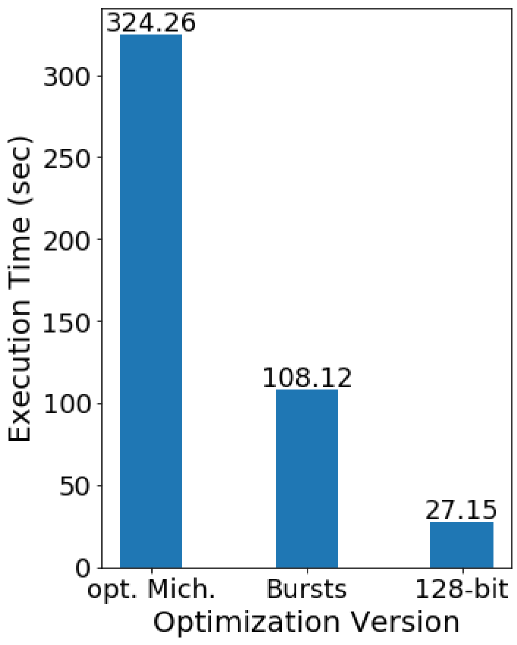


FIGURE 5.2: Execution time for various optimizations, in an FPGA deploying a single Michelsen accelerator core

An important point, which can also act as a guideline for anyone designing reconfigurable systems incorporating Xilinx's AXI implementations, is that the new approach not only allows four times more data to be sent to the accelerator

TABLE 5.2: Execution time and performance (sec and GFLOPS) for multiple Michelsen accelerator cores, properly deployed on a single FPGA, and clocked at 200 MHz

Time (sec)	Parallel accelerators	GFLOPS	speedup
27.15	1x accelerator slot	21.6	–
14.26	2x accelerator slots	40.9	1.9
9.84	3x accelerator slots	59.3	2.75
7.97	4x accelerator slots	73.3	3.39

due to the four times wider datapath, but also offers a reduced latency path that has to do with the AXI's disposal of all width conversion circuits in the path to the external memory. This reduced latency, now measured to about 250 ns, and resulted in a faster completion time for the issued AXI requests. We now measured a latency of 50cc at 200MHz and 76 clock cycle at 300MHz which give a latency of $(50cc \times \frac{1}{200}) \simeq (76cc \times \frac{1}{300}) \simeq 250ns$. This will become of even greater importance, once we have memory accesses generated by accelerators on remote FPGAs. This twofold effect, i.e. higher bandwidth & reduced latency, accounts for the established 4x improvement.

Subsequently, the next step was to employ more Michelsen accelerator cores within the same FPGA in order to process subsets of the original data set, introducing, therefore, in-parallel processing, e.g. two cores executing half of the 2400 blocks of 100K elements each. For the specific accelerator type, it has been possible to fit a maximum of four cores, utilizing 85% of the FPGA resources. Nonetheless, even at this high resource utilization, we could still reach a clock frequency of 200 MHz.

A related optimization while deploying multiple accelerators, pertains to the communication between the reconfigurable region of the MPSoC, i.e. the FPGA section where accelerators reside, and the DDR memory. Such accesses go through the Processing System (PS) of the MPSoC, and specifically through six ports that allow this. However, as two pairs of these links pass through single paths inside the PS, only 4 ports seem to be independent. Deploying these 4 independent ports and performing an analysis through various hardware accelerator configurations, has indeed been validated to produce the most efficient results. Minor deviations were also detected between coherent and non coherent ports, however not inducing any significant effect.

In order to exploit maximum available memory throughput, we properly used the UNILOGIC-related translation blocks, and through those we managed to assign a separate such port per accelerator. This allows for a maximum memory bandwidth exploitation. As investigated and presented in section 4.2.5 and through Figure 4.33, the aggregate memory bandwidth availability at the PS ports can reach about 9 to 10 GB/sec, and this can be achieved by deploying four PS ports in parallel.

Table 5.2 provides the performance results as we increase the number of cores, also plotted in Figure 5.3. The performance improvement with 4 accelerators running in-parallel reaches about 3.4x compared to a single accelerator. The deviation from the ideal 4x mainly appears as we move from 3 to 4 accelerators, as the maximum sustained memory bandwidth gets reached, revealing a memory bandwidth limit of the deployed MPSoC devices at about 9 GBytes/sec. The best performance we can achieve is 7.97 sec (equivalent to 73.3 GFLOPS), which is reported in [51] and is about 2.5 times better than our prior reported results in [75], demonstrating a significant improvement over our previous single-FPGA implementation of the UNILOGIC architecture.

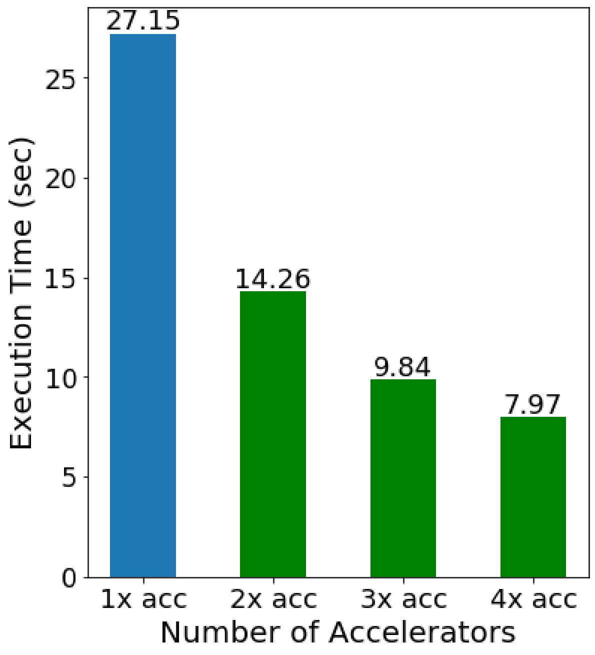


FIGURE 5.3: Execution time when properly deploying multiple Michelsen accelerators in a single FPGA

Finally, it is worth reporting that we were able to increase the clock frequency for various UNILOGIC architecture configurations. Designs were clocked up to the maximum allowed by the Xilinx tool flow for this FPGA technology, i.e. 333 MHz (regarding the PS-to-PL interface). We even performed some enhancements, that surpassed the tool limitations, through software configured, hardware clock managers on the configurable logic, making measurements for clock speeds up to 400 MHz. Successful implementation on such high frequencies was achieved by performing a full design space exploration of the High Level Synthesized accelerators [28, 123, 32, 82] and by hand-optimizing the HDL code of the scheduler-mailbox pair. Accelerator execution times were improved, near-linear in many cases. For example running a single Michelsen accelerator at 400 MHz resulted to about half the execution time compared to the 200 MHz execution. Deploying more such accelerators on the other hand, once again exposed and thus confirmed the memory bottleneck already revealed for the 200 MHz designs. However, running accelerators that are not so data hungry, would benefit from higher frequencies, as would be the case when mixing data-bound and computation-bound accelerators. Furthermore, the high frequencies reached guarantee that the UNILOGIC architecture would easily benefit from newer device versions, as vendors continuously introduce enhanced ones. More information on this investigation can be found in the following subsections, addressing earlier stages of our optimizations.

Moreover, we measured the performance of a design that does not support UNILOGIC and thus the Michelsen cores are directly connected to the Processing System of the device. In this way, we derived that the UNILOGIC architecture reduces the performance only by 3-4%, as the modest added latency of the UNILOGIC infrastructure is hidden by the aforementioned optimizations. We also measured that the latency, e.g. round trip time for a memory read access, introduced by the UNILOGIC architecture is just about 40 ns, which mainly corresponds to the latency of the central interconnect required by the UNILOGIC architecture. This is a small portion of the overall latency which adds up to 250 ns, which as stated above gets mainly introduced by the device specific latency of the Processing System interface and the access to memory. As will be presented in the multi-FPGA optimization and evaluation, UNILOGIC's latency comprises an even smaller portion of the more important cross-node round trip latency.

5.1.1 Earlier Optimizations for Single FPGA Acceleration

Before deploying the previously reported single FPGA optimizations, a lot of research and development effort has been invested, regarding aspects of the architecture such as HLS algorithmic optimizations, accelerator hosting logic and PS interfacing improvements (as parts of the UNILOGIC), along with bare metal testing and evaluation, with more details available in Appendix D. And preceding this phase, at the very early stages of this work, the architectural investigation and exploration of available devices took place, including the first Zynq Ultrascale+ device deployment through the first available commercial board by Trenz. This involved studying its features, interfaces, and available tools, and perform earlier test and evaluation. Then it was used to port existing architectures such as UNIMEM, as well encompass earlier components of the UNILOGIC architecture, while more details are available in Appendix C. The optimization aspects we want to present at this point include a wider variety of earlier configurations, so we have chosen to include part of those and arranged at this point as subsections. The reader can choose to bypass them, and consistently pass from the coherent single-FPGA optimizations reported above, on to the next section 5.1, that focuses on our investigation for remote acceleration.

5.1.1.1 HLS Algorithmic Optimizations

As we have mentioned above, the kernel originally implemented, is a variation of the Newton-Raphson method [1], specifically tuned for the Reservoir Simulation (RS) problem. The respective OpenCL kernel used in this work is "Michelsen" and provides a faster convergence in solving the Rachford-Rice equation for oil reservoir simulation. The initial implementation of the Michelsen kernel is not optimized for execution on FPGAs. Hence, manual code transformations have been applied in order to reach efficient hardware implementations. In the Michelsen case, at each grid point, the original OpenCL code is sequential due to a *while* loop, and the aim of the optimizations has been to overcome this obstacle and, therefore, introduce and maximize parallelism. The main work related to this thesis was the co-design of the accelerator and hosting hardware, as well as continuous and incremental hardware deployment, testing and evaluation.

In addition, a set of directives have been manually introduced for further optimization benefits. This work has been heavily based on Xilinx's Vivado HLS [119]. This is a tool that produces equivalent RTL designs [111] to high-level model descriptions of various algorithms. The Vivado HLS tool offers a set of *Directives* in order to make the respective codes more efficient for FPGA implementation. Therefore, the process has been to start off with an initial set of directives as well as code modifications introduced manually that improve the performance of the OpenCL kernels. Consequently, further exploration steps have been performed automatically using the ECOSCALE *Design Space Exploration* [58] (DSE) tool, which has been developed by Politecnico di Torino, and our joined efforts on its exploitation are reported in [75]. The latter has been used to enable automated design space exploration and micro-architecture definition without requiring the designer to specifically know the architectural features of the underlying FPGA. The DSE tool provided an additional set of directives for the OpenCL kernels of the manual stage and through the merging of the two, came the final version used in the production of the accelerator core.

The manual analysis consisted of carefully investigating the Michelsen OpenCL kernel. The main characteristic of the algorithm to take advantage of, is that of multiple small *optimization* problems, which are independent of one another. A suitable way for doing this is by pipelining the process.

Algorithm 1 Initial Reservoir Simulation (RS) pseudocode

```
float a, b
for (all grid points)
    /*initialization*/
    a=amin
    b=bmin
    operation 3
    ...
    while (optimization target)
        b=b+1
        a=a+1
        operation 3
        operation 4
        ...
    end while
    operation 1
    ...
    write result
end for
```

The original OpenCL pseudocode of the target application looks like the one shown in Algorithm 1. The problem here is that this form of the code description does not allow for the use of a pipeline directive due to the *while* loop, otherwise called *optimization* loop since the coefficient it produces approximates, to an acceptable level of deviation, a real-life value. This loop is burdened with dependencies as well as being non-statically bounded, thereby making it a sequential process that cannot be pipelined. This would make the FPGA implementation of such algorithms inefficient due to the inherently slow FPGA clock frequency, which results in significant slow downs compared to the equivalent software running on a conventional processor.

Hence, we propose a method for making such cases efficient for implementation on FPGAs. The HLS tools cannot pipeline non-statically bounded *while* loops but they can pipeline statically bounded loops very efficiently. A restriction is that the small optimization problems have to be independent from one another. Therefore, we had to move the *while* loop to the top of the code and the *for* loop inside it. As such the tool can pipeline the latter which results in significant speedups. Consequently, the pseudocode that represents the modified architecture is shown in Algorithm 2. The two architectures, i.e. *initial* and *optimized* are also graphically depicted in Figures 5.4 and 5.5.

Algorithm 2 Optimized Reservoir Simulation (RS) pseudocode

```
/*All variables become arrays*/
int a[pipeline_size]
int b[pipeline_size]
while (all problems solved)
    for (n : all grid points)
        if (initialization)
            /*initialization*/
            a[n]=amin
            b[n]=bmin
            operation 3
            ...
        end if
        else /*while loop code*/
            b[n]=b[n]+1
            a[n]=a[n]+1
            operation 3
            operation 4
            ...
            If (optimization reached)
                operation 1
                ...
                write result[n]
                    solved++
            end if
        end else
    end for
end while
```

The improvement of such an approach is that the hardware that would remain unused while each loop is executing is now utilized by another optimization problem. Each independent problem will have exactly the same amount of iterations inside the *while* loop as it had with the previous code.

Unrolling the *for* loop at the top was also investigated, however, no significant benefits were obtained, hence, this option was rejected. That is because it led to a small speedup at a cost of significantly high area overhead.

A significant optimization on the kernel compared to the ones presented in [75] was the usage of vector types (float4) for the I/O in order to support 128bit, a feature co-designed with and supported by the UNILOGIC platform. This, along with the modification of the code segments that perform the I/O in order to invoke 256 word bursts provided the significant reduction of the I/O overhead, as was presented above.

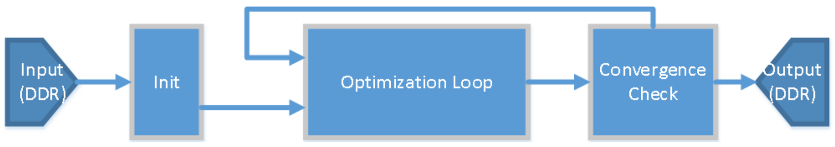


FIGURE 5.4: Block diagram of initial architecture

For the initial kernel, a manual set of directives was chosen as the appropriate directives were obvious. The main calculations are done inside the unbounded *while* loop body which cannot take any performance oriented directive. The only meaningful performance improvement was achieved by partitioning the temporary input tables which allowed pipelining the smaller *for* loops over the components in the code, including those inside the *while* loop.

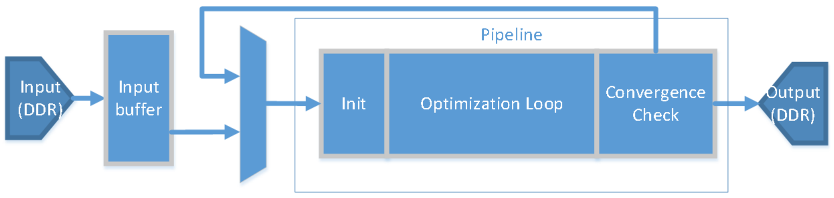


FIGURE 5.5: Block diagram of optimized architecture

5.1.1.2 Earlier UNILOGIC & accelerator co-design enhancements

The first evaluation results, that mainly proved the feasibility of our approach, were based on a single-FPGA board and they are presented in [75]. In this work, we mainly reported execution metrics for two accelerators implementing the Michelsen and Hyperbolic algorithms, which are utilized in the calculation of the Rachford-Rice [124] equation. The main outcome was that the UNILOGIC/UNIMEM architecture can be implemented; even this very simple implementation on a single FPGA triggered a 30% speedup of a multi-core CPU and a 2x to 13x better energy consumption.

Moving to the multi-FPGA QFDB boards and the multi-node system, the efficiency of the intercommunication between the FPGAs, within the same and/or in remote nodes, and the remote deployment became a matter of principal importance.

By analyzing the performance of the accelerator in our QFDB platform, we realized that the problem was mainly I/O bound since the bottleneck was on the AXI interconnect (which is connected to the memory controller) since its standard version does not support more than a mere of two pending transactions. As we already reported, in order to overcome this bottleneck we first incorporated burst transactions and then we employed a wider datapath. Then we upgraded the accelerators and all the design blocks in its datapath to match the maximum bus-width supported by the AXI implementation on the new FPGAs, i.e. 128-bits while still supporting bursts. The plots and tables on the 200 MHz designs, were reported above in section 5.1.

Table 5.3: Execution time (ms) for a single semi-optimized (pipeline 2) Michelsen core in various configurations

Time (ms)	**Optimization Version**	**Frequency (MHz)**
300	No Optimization	125
187	No Optimization	200
100	32-bit with bursts	125
29.5	128-bit with bursts	100
25.0	-//-	125
17.4	-//-	200
15.3	-//-	250
13.8	-//-	300
13.3	-//-	333

However, looking in Table 5.3 and Figure 5.6 we can see this this improvement on our earlier designs reaching at that time 125 MHz, while in Figure 5.7 and on the remaining rows of Table 5.3, we can see how the execution time dropped as

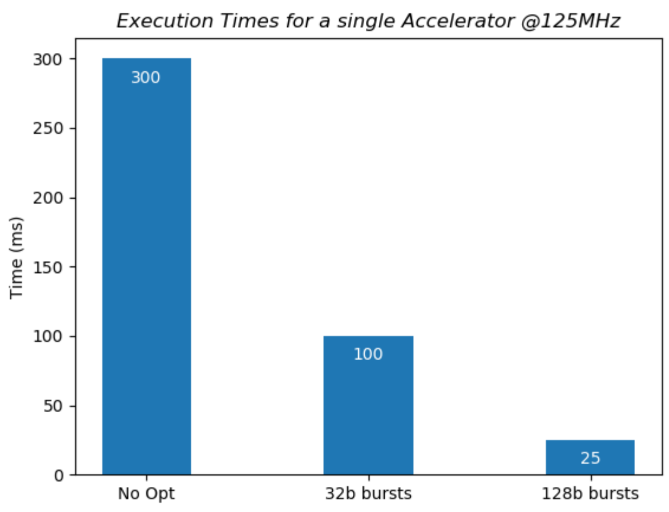


FIGURE 5.6: Optimizations on early UNILOGIC designs, operating at 125 MHz

we improved the design's clock frequency. It even reaches the 333MHz frequency barrier for a single clock domain design, as at that point the FGPA was not congested, reaching a utilization of about 72%. The 333 MHz frequency is the upper limit allowed by the Xilinx tools for the MPSoC ports.

Furthermore, we added more accelerators running in parallel. Here we also included results with higher frequencies, as be seen in Table 5.4 and Figure 5.8. Two different versions were implemented and tested. One has an AXI interconnect between the accelerators and the MPSoC (and hence the DRAM memory). The other does not, but instead the accelerators are connected directly to the MPSoC ports. The second version, which triggers better performance, was implemented and evaluated at two clock frequencies. At this point, moderate FPGA congestion due to higher resource utilization, allowed clock frequency to reach 300 MHz.

TABLE 5.4: Execution time (ms) for muplitple semi-optimized (pipeline=2) Michelsen cores running in a single FPGA

1x ACC	**2x ACC**	**3x ACC**	**4x ACC**	**Freq. (MHz)**	**Ver.**
15.3	10.4	9.7	9.3	250	AXI
11.6	6.9	6.1	5.9	250	no AXI
9.9	6.4	6.0	**5.8**	300	no AXI

Observing the results, we should note that the intervening AXI interconnect hindered performance. Even with wide 128 bit buses and bursts, the latency of

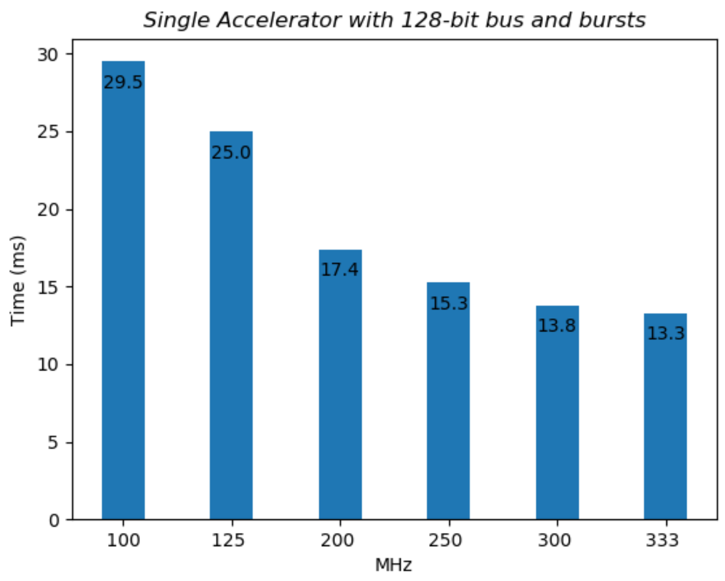


FIGURE 5.7: Improving clock frequency on the optimized design

the path to the memory, although quite low, again does not allow the path to be saturated and fully utilized. This is the reason we get these differences in execution times. To make sure it was this case, we again turned to observing the hardware execution. Indeed, the latency of the non AXI version, dropped from 250ms down to 210ms. Chipscope assisted measurement showed 42cc @200MHz ($42cc \times \frac{1}{200} \simeq 210ns$). As for the other two cases of Table 5.4, we should first note that the value in bold, namely the 5.8ms, is the lowest execution time and quite lower that the **9.3** ms achieved at the quad-accelerator version of our prior Trenz platform reported in [75]. This corresponds to a 60% improvement, which also translated to a 2 times faster execution than the 4-threaded CPU run. Another information worth noticing on this table, is that execution speed reaches a boundary somewhere around the 5.8ms to 6ms mark, and the improvement in frequency cannot offer much once we come near this. This actually reveals a point where the DDR path throughput gets saturated. The best performing software execution in the CPU and then the corresponding best performing hardware execution in the FPGA, can be seen in Tables 5.5 and 5.6.

This saturation was then investigated, and as reported through our latest evaluation reported above, it corresponds to the 9 GBytes per second maximum memory throughput, bottlenecked by the PL-to-PS and PS-to-memory interfacing. What is more, by later incorporating more pending AXI transactions for local access

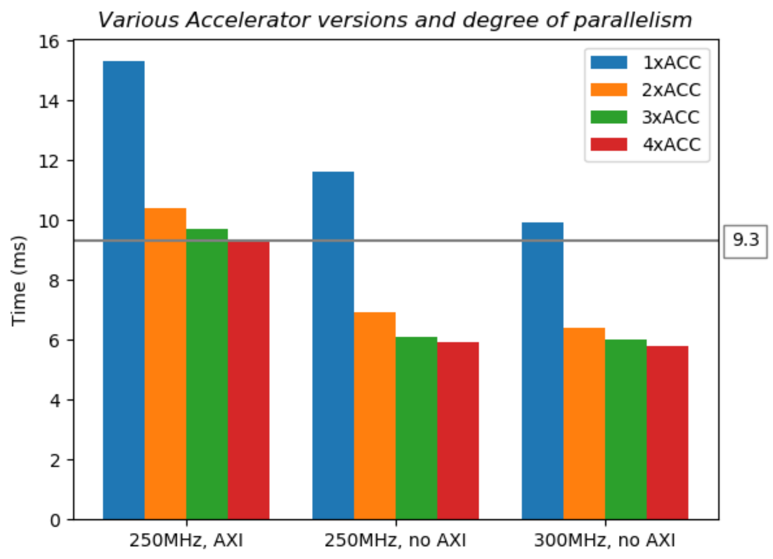


FIGURE 5.8: Results for multiple accelerator cores, running in parallel, in a single FPGA, either with an intervening AXI interconnect or directly connected to the MPSoC ports

TABLE 5.5: Best software execution times (ms) form single- and four-threaded Software (CPU) execution (OpenMP)

	CPU 1 thread		**CPU 4 threads**	
Data size	100K	200K	100K	200K
Hyperbolic	25.5	50	8.8	17.2
Michelsen	29	56	11.7	22.3

paths as well, diminished the negative effect of the intervening AXI that was seen through Figure 5.4.

TABLE 5.6: Best hardware execution times (ms) for simple (non-optimized) and optimized Michelsen Hardware Accelerators (FPGA)

Data size	**100K**	**200K**	**SpeedUp vs. SW**
Simple 8-core	117	235	0.1
Opt. 4-core AXI	9.3	18	1.2
Opt. 4-core no AXI	5.8	11.2	2

Importantly, at that time, in order to perform a design space exploration on the Michelsen to be used as a reconfigurable accelerator, we have built two versions of it: one with a deep pipeline and another one with a swallow one; the latter, simpler version, utilizes *40K LUTs (14.5%), 65K CLB regs (12%), 8.7K CLBs (25%) and 176 DSPs(7%)*, while the former utilizes *69K LUT (25%), 105K CLB reg*

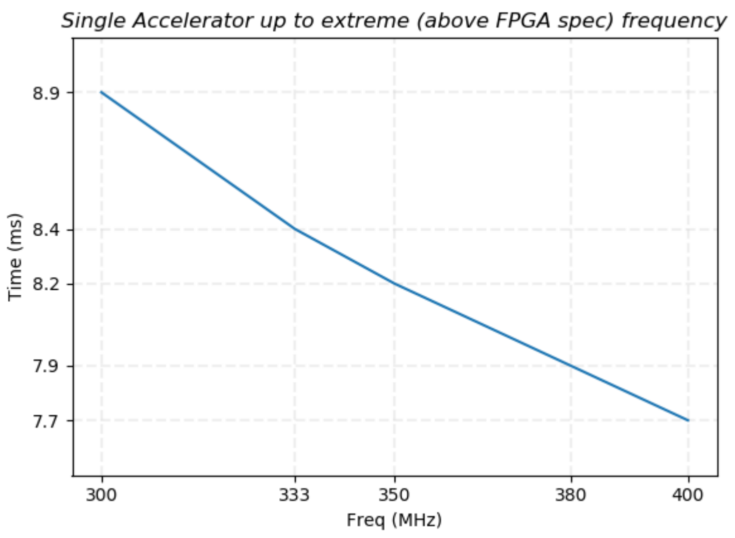


FIGURE 5.9: Fully optimized accelerator (pipeline=1) execution at various frequencies

(19%), 12K CLB (35%), 307 DSPs(12%). Although the performance-optimized one consumes about 70% more resources than the resource-optimized version (in terms of LUTs which seem to be the crucial resource for the complete FPGA design), it can still be utilized in our system, by combining two reconfigurable slots. However, in our performance exploration we opted for the use of the resource-optimized version since it covers only one slot and its performance is more than 50% of the performance of the fully optimized one. This is important, when we think of a design that targets partial reconfiguration, and does not pertain to the usual flow that gives an inflexible, static FPGA design. Furthermore, this lighter version also gives us more freedom in testing a variety of parallel execution patterns.

TABLE 5.7: Extreme case measurements with fully (performance) optimized Michelsen core (pipeline=1) to test accelerator limits and investigate on DDR path saturation

Time (ms)	**Interconnection**	**Freq. (MHz)**
12.5	Managed AXI	333
8.9	Straight to PS port	300
8.4	-//-	333
8.2	-//-	350
7.9	-//-	380
7.7	-//-	400

Still the deep pipeline version provided a useful aspect on the design exploration, as this modules can test the saturation throughput of a single MPSoC port. The results, on this aspect, can be seen through Table 5.7, and Figure 5.10. The execution time is decreased almost linearly with the frequency increase. Note that in the tables we report those numbers as extreme cases. This has mainly to do with the clock frequency and the fact that we surpassed the 333 MHz MPSoC port restriction. In order to do so, a clock wizard was added in the designs, inside the reconfiguration logic. This is intervening between the PL clock generated by the MPSoC and the clock signal that controls the main FPGA logic. We added a configuration AXI lite port to this clock generation module (aka clock wizard), in order to be able to alter the clock frequency after the FPGA fabric is configured and running. So we initially synthesize and place and route the design with the design tool's maximum allowed frequency, which is 333 MHz for our devices. Then we alter the clock frequency of the accelerator in runtime, configuring appropriately the clock wizard through the AXI lite port, through processor simple commands. The clock can thus be hot-switched to any desired frequency value. We moved on with increasing it, up to the point we can still have the hardware running sound, providing fully proven results and no error indication. By applying this method we were able to successfully clock the accelerator up to 400MHz.

Table 5.8: Extreme case measurements (execution time in ms) with 1x and 2x fully opt Michelsen cores (pipeline=1)

Freq. (MHz)	**1x ACC**	**2x ACC**
300	8.9	6.4
333	8.4	6.2
350	8.2	6.1

Moving to Table 5.8, we can see the results of our next design exploration efforts, which include deploying two fully optimized cores, running in parallel on a single FPGA. Figure 5.10 plots these results, along with the single accelerator ones described above. Even though the performance improves, the improvement is not linear as to the number of accelerators. This has to do with the memory communication saturation point around the 6ms mark, which has been described above. Also it verifies our assumption, that the resource-optimized version of the accelerator, can be more convenient than the performance-optimized one for design space exploration purposes; the former can give fully utilized 2x, 3x, or

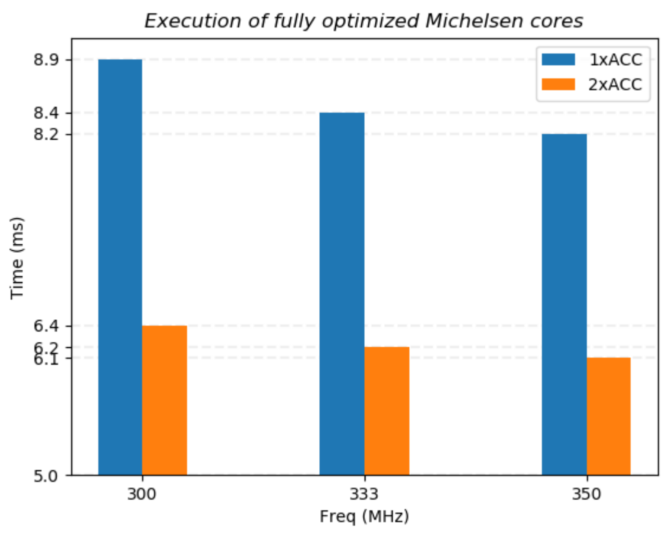


FIGURE 5.10: Two fully optimized cores (pipeline=1) running in parallel on a single FPGA

4x design versions. On the other hand, the performance-optimized version proves how efficient we can get through a single accelerator, connected to a single MPSoC port. So if our limiting factor is the MPSoC ports, we can use less but performance optimized accelerator versions, in a 'single accelerator per port' approach.

5.2 Optimizations for remote (cross-FPGA) acceleration

An important milestone on the way to offer an FPGA unification platform, where parallel accelerator tasks can run either locally or remotely in an effective manner, is that of reaching a close match between local and remote execution times. Remote in the sense of invoking an accelerator in a remote FPGA, which in turn will have to operate on data that reside remotely. Using our data hungry, optimized Michelsen accelerator core, makes this milestone harder to reach, but provides a more sound proof. Even non perfect matches between local and remote execution, would indicate that remote accelerator invocation is a viable alternative. For example, a 30% degraded remote performance, would signify a benefit from invoking an already configured remote accelerator slot, or, in a similar case, a pair of remote accelerators. However, as we will describe below, we further opted for the closest match possible.

Not encompassing any specific optimizations for remote accelerator scenarios, proves to greatly degrade performance. Using the same data sizes, and design frequencies for ease of comparison, we measured the Michelsen accelerator execution to increase from 27 sec in the local execution scenario, to a discouraging 119 sec when remote invocation and remote data usage takes place. This happens as latency now grows substantially, advancing from one node to the other, passing through the GTH links and their surrounding logic, as well as through the additional AXI interconnect on the remote node. Paired with AXI's poor count of allowable pending transactions, it leads to significant stalls while waiting for a response to arrive, and before issuing subsequent requests. In order to confront such an inefficiency, a series of optimizations had to be examined and incorporated. We report here on two main optimizations, while we also present our research on two more critical aspects, that can further fortify the UNILOGIC approach.

5.2.1 Hiding latency

More pending transactions had to be allowed, as this can compensate for the added latency presented in the inter-FPGA path. The round trip, measured with hardware monitoring through the Xilinx Chipscope tool flow, reached an impressively low 240 cc @200MHz, thus ($240cc \times \frac{1}{200} \simeq 1200ns$). This is quite lower than what we have seen reported by other multi-FPGA implementations. Nonetheless, this is still 950 ns higher than the 250 ns latency presented in the local accelerator path. In order to hide this added latency through ongoing transactions, we switched to a completely manual AXI interconnect configuration. For this, Xilinx gives the option to enable manual interconnection configuration, referred to as *"un-managed"*. This non reversible process, halts any automatic configuration offered (such as address mappings, automatically adding and removing slaves and masters, using width and protocol converters and many other), and everything is left to the designer's discretion. Most importantly, a positive outcome is that through this added manageability the pending transaction availability is affected, which can now be substantially raised. The actual number depends on various interconnect and related design parameters, and in our case the design parameters allowed for the maximum allowance that can be offered, i.e. 32 pending transactions. These ongoing -mostly burst- transactions add up to a total amount of time that is quite higher than the round trip time, thus eliminating stalls. This approach triggered a significant performance increase, as recorded in row 2 of Table 5.9 and plotted in Figure 5.11. The remote execution time now drops impressively from 119 sec down to 35 sec. This renders the local execution comparable, outperforming remote execution by 32%.

Table 5.9: Execution time for remote FPGA accelerator utilization. A Michelsen core with 128-bit bus and bursts is used

Local (sec)	**Remote (sec)**	**Central AXI Version**	**GTH link**
27.15	77	default AXI	10.3 Gbps
27.15	23	un-managed/max-Pending AXI	10.3 Gbps
27.15	**28.11**	un-managed/max-Pending AXI	16.25 Gbps

5.2.1.1 An Analysis on Latency, Hop Count & Bursts

It is useful to devise a formal mathematical representation of the latency introduced per node traversed, and in parallel of the burst size and burst count that needs to be supported so that this latency can be hidden. If the metrics to be calculated are met, then depending on the accelerated algorithm, by encompassing a proper data fetch mechanism we should be able to absorb the introduced latency. In our implementation, that deploys the AXI protocol and interconnects, a burst can reach a maximum size of 4 KB. With the data bus at its maximum allowed width of 128 bits (16 Bytes), and with a clock cycle (cc) of 5 nanoseconds (ns) for our 200 MHz designs, the burst duration, BD(1), equals:

$$BD(1) = \frac{BURST\_SIZE}{BUS\_WIDTH}$$

which for the implementation sizes mentioned above accounts for $BD(1) = \frac{4096}{16} = 256\ cc$ and in nanoseconds $BD(1) = 256 * 5\ ns = 1280\ ns$. In order to further examine the amount of latency that can be hidden by a varying number of bursts, we should determine the multi-burst duration. Since the single burst duration, BD(1), is 1280 ns, for a number of 'b' back to back bursts we get a total burst duration, BD(b), equal to:

$$BD(b) = \frac{BURST\_SIZE}{BUS\_WIDTH} \times b \qquad (5.1)$$

which actually results to $BD(b) = BD(1) \times b = 1280ns \times b$

Then, to properly formalize latency, we should firstly focus on the following latency related designations:

- DDR access from PL (**$\mathbf{DDR_L}$**): 250 ns from data request to data response, which is independent of the PL clock. This equals to 50cc @200MHz
- C2C latency (**$\mathbf{C2C_L}$**): 220 ns or 44cc @200MHz, applicable at any FPGA boundary
- RTT between two FPGAs (**RTT(1)**): 1200 ns or 240cc @200MHz. This is the Round Trip Time for a remote FPGA access, including DDR access and the four related FPGA boundaries, thus four times the C2C latency
- Intra-FPGA latency (**$\mathbf{INTRA_L}$**) is the time spent in the intra-FPGA circuits, which however are minimal, as the UNILOGIC related translation circuits operate on the fly, adding no extra clock cycles to the paths they intervene. This will be calculated below, and includes both forward (request phase) and backward (response phase) traversing of the FPGA logic

Now, if RTT(1) is the Round Trip Time to travel 1 hop away, and **RTT(h)** corresponds to traversing 'h' hops, we get:

$$RTT(1) = C2C_L \times 4 + DDR_L + INTRA_L \times 2 \tag{5.2}$$

Since the above latency is was measured on the prototype to reach a total of 1200ns, while also verified by the partial measurements that resulted to the numbers reported above, we get:

$$RTT(1) = 220ns \times 4 + 250 + INTRA_L \times 2 = 1200ns$$

This means that the $INTRA_L$ accounts for 35 ns per FPGA, which as mentioned accounts both for forward and backward FPGA traversal. Moving on to the generalized RTT(h) type, we get:

$$RTT(h) = C2C_L \times 4 \times h + DDR_L + INTRA_L \times (h + 1) \tag{5.3}$$

or in nanoseconds (ns):

$$RTT(h) = 220 \times 4 \times h + 250 + 35 \times (h + 1)$$

Now if we want to calculate the minimum number of bursts required to hide the latency for a number of 'h' traversed hops, or symmetrically, to find the maximum hop count that can be traversed for a specific implementation that supports a number of 'b' bursts, we should rely on the following equation:

$$RTT(h) = BD(b) \implies$$
$$C2C_L \times 4 \times h + DDR_L + INTRA_L \times (h+1) = \frac{BURST\_SIZE}{BUS\_WIDTH} \times b \tag{5.4}$$

and using our implementation specific numbers, we get:

$$RTT(h) = BD(b) \implies 220 \times 4 \times h + 250 + 35 \times (h+1) = b \times 1280$$

Then solving for 'h' or for 'b', one gets the maximum hop count, or the minimum burst number for a specific implementation. As a short reference, we include a small number of representative examples in Table 5.10.

Table 5.10: Correspondence between hop count and number of bursts

Hop Count (h)	No. of Bursts (b)
1	1
4	3
8	6
11	8
15	11
44	32

Elaborating on our implementation of the Michelsen accelerator as a use case, every data read phase includes 2048 elements, of 16 4-Byte float numbers. This results to a data transfer size of $2048 \times 16 \times 4\ Bytes = 128\ KB$. This size is equal to $32 \times 4\ KB$ or in other words it gets transferred through 32 AXI bursts[2]. With 32 bursts we could hide a round trip latency corresponding up to 44 hops. Taking into account that our 64-FPGA prototype has a max distance of 3 hops, the performance of this accelerator implementation could scale to much higher platform sizes.

[2]In our final implementation, each 4 KB AXI burst is fulfilled through 16 bursts of 16 cycles each. 16 cycles on a 128 bit bus transfer 256 Bytes, so 16 small bursts result to the 4 KB burst

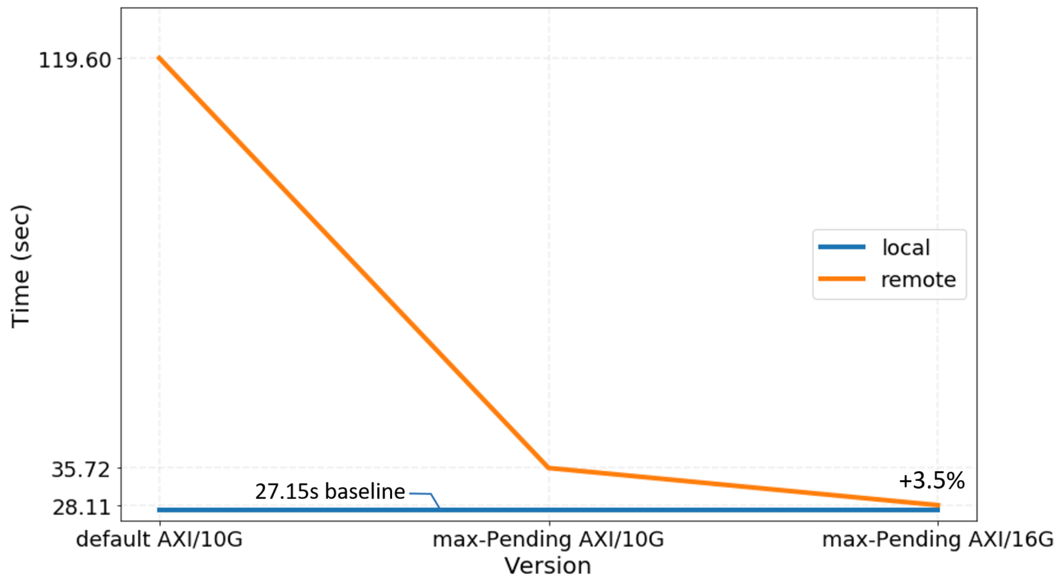


FIGURE 5.11: Accelerator execution times for local and remote FPGA runs. A remote run includes remote accelerator configuration, and then remote memory usage. Almost similar local and remote execution time is achieved

5.2.2 Supporting higher bandwidths

Further investigating, and monitoring the actual execution patterns on the real reconfigurable hardware, we noticed that the bottleneck of the design has now been shifted from the AXI interconnect, to the inter-FPGA links, which were at that time set to 10.3 Gbps. Such speed is mostly supported by both SFP+ connectors and copper cabling as well as board traces connecting FPGAs on the same board. It is also the maximum operational speed on the commercial platforms we have utilized in the initial phases of validation. To further improve this, while making good use of the highly efficient QFDB board design, we designed an improved version of the chip2chip (C2C) module. This utilizes the maximum FPGA transceiver speed supported by the current devices, reaching 16.3 Gbps. Deploying this new C2C module led to significantly reducing remote acceleration time, as seen in the 3rd row of Table 5.9. Time to solution now reaches 28.11 sec, which now brings the remote acceleration almost on a par with the local one, now getting outperformed by a marginal 3.5%. In other words, remote accelerator delivers almost the same performance, which greatly consolidates the UNILOGIC architecture in its principal aim to offer efficiently unified, reconfigurable resources in an effective, virtualized environment.

5.2.3 Further reducing latency

In parallel, we continued our efforts in order to achieve lower latency. On this directions, we targeted on the chip2chip (C2C) IP block, as it contributes a main portion of the inter-FPGA latency. The C2C which embodies the Xilinx's Aurora IP as presented in section 4.2.6, introduces a latency of about $220ns$, which is appended at a path every time it crosses a chip boundary, i.e. at every FPGA boundary. This means that it gets added four times in a cross-chip round trip time (RTT). We replaced the Aurora IP with a custom, low-latency Serialization-Deserialization (SerDes) IP block, that directly controls the GTH transceivers. This was measured to offer substantially reduced latency, now falling to $90ns$ at each FPGA boundary. Our efforts on this custom C2C module have been detailed above in section 4.2.6.1. This reduction is reflected to the RTT, which dropped from the $1200ns$ reported above, down to an impressive $680ns$. Although even the previous $1200ns$ of latency allowed a near-match of local to remote acceleration, this significantly reduced latency of $680ns$ allows for a higher number of intermediate nodes (hops) to be traversed, e.g. from an accelerator to the remote memory it accesses, with no considerable performance degradation. What is more, the initial setup of a remote accelerator comes at almost zero cost, especially compared to the time it will spend for the designated job. Real hardware tests on remote acceleration tasks revealed that, even when the maximum distance of our implemented topology had to be traversed (i.e. three FPGAs away), no significant performance degradation occurs.

5.2.3.1 Updating the Hop Count Analysis under Reduced Latency

TABLE 5.11: Updated results for hop & burst count correspondence, deploying the custom C2C that reduces latency to 90 ns

Hop Count (h)	No. of Bursts (b)
1	0.5
6	2
9	3
12	4
15	5
103	32

If we want to update our analysis based on the custom C2C latency, we should use this reduced latency to recalculate equation (5.4), which now results to:

$$RTT(h) = BD(b) \implies 90 \times 4 \times h + 250 + 35 \times (h + 1) = b \times 1280$$

If we solve this equation either for the number of hops 'h', or for the number of bursts 'b' we get highly improved results. Table 5.11 now gives the updated representative examples of Table 5.10, for the same or similar hop count.

5.2.4 Further enhancing connectivity

As data transfer, or equally throughput availability, is a common cause of bottlenecks, additional research that targets optimizations for inter-FPGA communication throughput is beneficial. A favorable approach is to further enhance the C2C IP block by incorporating link bonding, as presented in 4.2.6. This way, pairs of transceivers are presented through a single AXI interface to the rest of the FPGA logic, giving the notion of a single link of double the throughput. For the GTH transceivers this means $2 \times 16.3Gbps = 32.6Gbps$ of throughput for intra-QFDB communication. Such a higher bandwidth availability relaxes the link's marginal capacity, and allows our system to efficiently support applications that require very high bandwidths between Workers. For inter-QFDB communication, link bonding translates to $2 \times 10.3125Gbps = 20.625Gbps$, contributing for a similar effect within larger "neighborhoods" of Workers.

We would additionally like remind the aforementioned state-of-the-art improvements on available FPGA transceiver bandwidths which have currently reached up to 58 Gbps or even greater, as in the case of Intel FPGA incorporating transceivers up to the 112 Gbps mark. Furthermore, a variety of FPGA or MPSoC flavors, from either Xilinx or other vendors, incorporate DDR controllers that offer much higher memory bandwidths, while many of these controllers can be encompassed per FPGA. This manifests that the memory bandwidth domain, is not to be considered as a typical disadvantage of the reconfigurable technology field. Importantly, the UNILOGIC approach and techniques can gracefully be deployed to maximize utilization of available capabilities in any state-of-the-art platform alternative.

Chapter 6

Full System Evaluation

In this section we evaluate the UNILOGIC architecture on the QFDB-based prototype described in Section 4.1. We progressively deploy and evaluate the architecture, introducing the hardware accelerators initially on a single FPGA and eventually on more FPGAs on a single or multiple QFDBs of our prototype. Then, reaching to the final evaluation step, our architecture is deployed on all QFDBs of our platform, allowing acceleration tasks to be spread and running in parallel on all 64 FPGAs of the prototype. Depending on the accelerator characteristics, proper fine-tuning for compute intensive accelerators, or appropriate data distribution for data intensive accelerators, allows the architecture to scale efficiently. Therefore two opposite cases of algorithms, and thus two resulting diverse accelerators, are reported, providing the means for a comprehensive UNILOGIC evaluation. Both algorithms are based on third party software, in order to fulfill the needs of real life computational loads. This means that the evaluation is not based on synthetic, possibly convenient workloads, thus giving higher credibility to the results presented. The first case concerns the aforementioned Michelsen algorithm, which after a thorough flow of optimizations, leads to a data-intensive accelerator implementation. In the second case, deploying the KNP algorithm results in a compute-intensive accelerator. This KNP accelerator is based on the Lukas Kanade's optical flow algorithm, widely used in monitoring road traffic, with more deployment details presented in section 6.3.

In the same manner as in Chapter 5, I devised the whole platform hardware, with the KNP HLS code contributed by Synelixis, Acciona and Politecnico di Torino (PoliTo). I set up the whole FPGA firmware which implements the complete UNILOGIC architecture through its Worker building block. Also I explored on the execution scenarios for the platform-wide testing, and the testing itself, while also devised the architecture for remote access to the platform. The CPU and GPU porting and executions for the Michelsen and the KNP accelerators was taken care by Synelixis and Acciona/PoliTo respectively.

6.1 Case 1: Evaluation with the Data-Intensive Michelsen Accelerator Core

In order to evaluate how the UNILOGIC approach scales, we ported our architecture on the final prototype, incorporating 64 FPGAs, as presented in Section 4.1.5. The UNILOGIC architecture is eventually implemented on all the FPGAs, allowing for reconfigurable resource unification and promoting parallel execution of distributed accelerators. The advantage of UNILOGIC is that it offers and utilizes the resources of many FPGAs in a way that they are all treated as one, unified, parallel-processing platform. Hence, it is important to evaluate the performance of UNILOGIC when one such parallelizable application is to be processed.

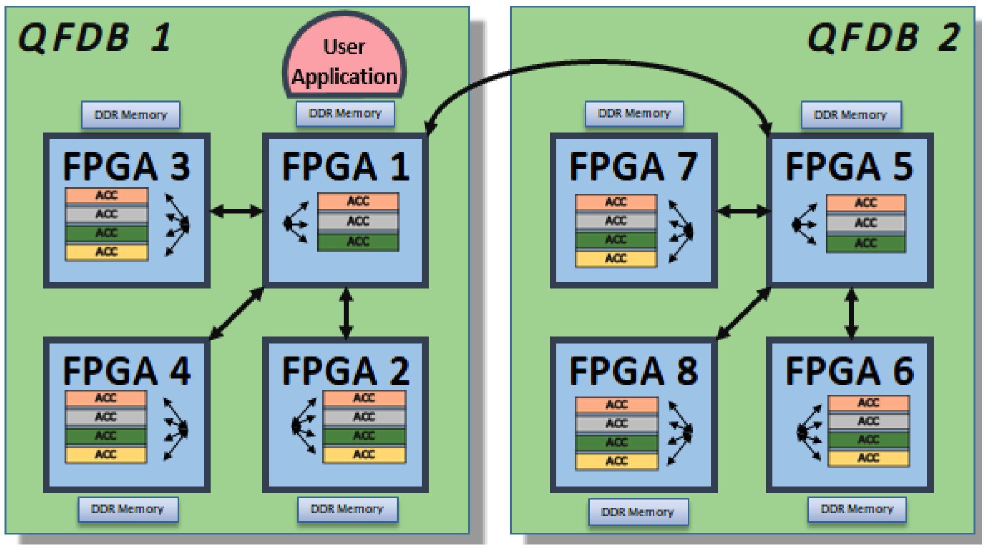


FIGURE 6.1: Michelsen accelerator distribution for 2 interconnected QFDBs

For this test, the Michelsen accelerator core has been used. Each QFDB hosts 4 FPGAs and one of them, which we have designated above as the "Network" FPGA, is loaded with added hardware functionality, compared to the other three. This, as reported before, is owed to the networking infrastructure it hosts, in order to cooperate with other QFDBs. This limits the space available for accelerator core instantiation, thereby, allowing for a total of three. Therefore, as many as 240 hardware instances have been deployed across the evaluation platform that consists of 64 FPGAs; 48 FPGAs hosting 4 Michelsen accelerator cores while the remaining 16 hosting 3. This is due to the particular evaluation setup that consists of a platform that uses 16 QFDBs.

Figure 6.1 shows the accelerator core distribution across a simple setup of two QFDBs. Hence, FPGAs 1 and 5 host three in contrast to the remaining FPGAs that can host four. Inter-QFDB communication takes place via FPGAs 1 and 5, which is why there is less space available for accelerator hosting. Note that the accelerator concentrations (3 and 4) are a function of the algorithm that they realize, i.e. Michelsen. Simpler/more complex algorithms lead to accelerators with smaller/greater hardware imprint, hence, more/less can be hosted within the same FPGA.

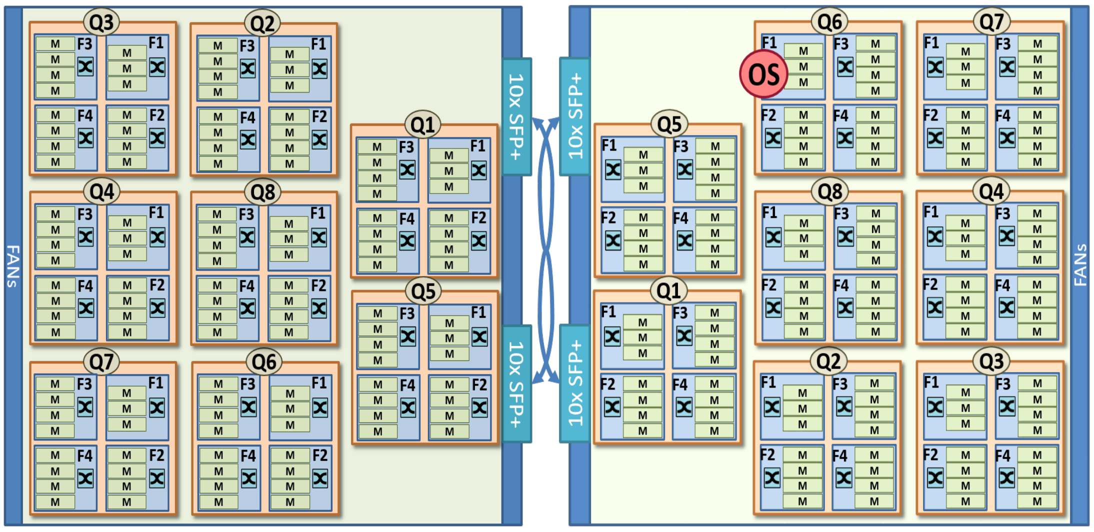


FIGURE 6.2: 240 Michelsen accelerator cores (green boxes) and a single OS (red circle) in 64 FPGAs

Subsequently, Figure 6.2 shows the entire many-accelerator evaluation setup. Specifically, it consists of two interconnected baseboards, each hosting 8 QFDBs, Figure 4.16, i.e. 16 in total. Hence, Figure 6.2, shows the 240 Michelsen accelerator core distribution, depicted in green "M" labeled boxes within a pair of baseboards, and with the QFDBs interconnected in the hypercube topology of Figure 4.9. Each baseboard hosts 120 Michelsen accelerators and the two baseboards are connected together via the two SFP+ cables in the middle of Figure 6.2. Finally, for one of the two baseboards, a Linux Operating System has been hosted on the ARM

TABLE 6.1: Execution times (sec) using 1 vs 240 in-parallel accelerator cores

	1 accel. core	**240 accel. cores**
Execution time (sec)	27.15	0.179

processors of FPGA 1 of Q6, depicted in a red circle, to facilitate the experiment, i.e. user-interfacing.

Furthermore, owed to its parallelizable nature, the complete application task has been separated into 2400 sub-tasks that can be distributed across the evaluation platform in different combinations. Each accelerator core can tackle one sub-task at a time. Hence, at the worst, the total workload can be served using a single accelerator core, i.e. fully serialized implementation, or, at best, in ten groups of 240 in-parallel executions. Each task at both its initialization phase as well as the subsequent completion phase, entails light synchronization, which slightly affects overall execution time, due to the beneficial implementation's low latency attribute. The aforementioned examples are the two extremes of the evaluation scenario and are shown along with their corresponding completion times in Table 6.1. An initial observation is that with full accelerator core utilization, a 152x speedup over the equivalent in-series execution has been measured.

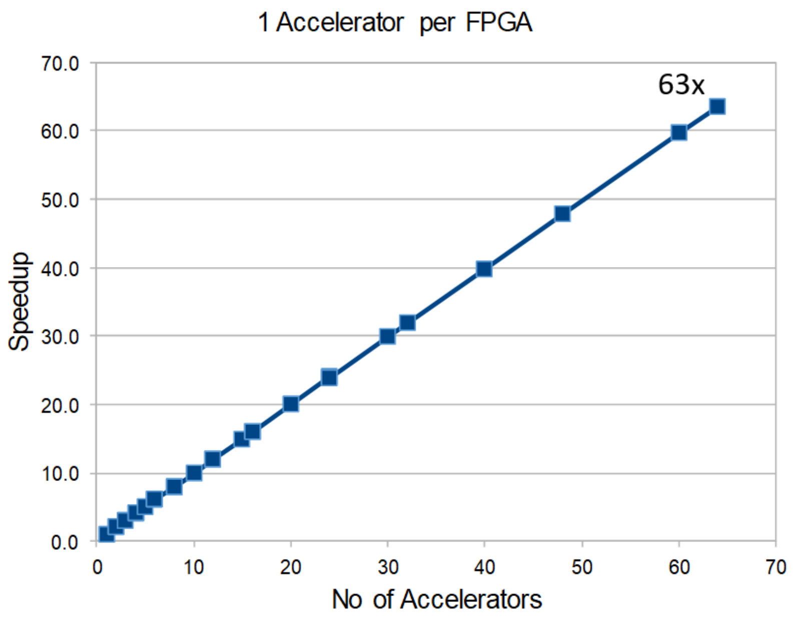


FIGURE 6.3: Speedup vs no. of accelerators in the case of a single accelerator per FPGA

In-between those two figures, lays a set of measurements that shows the platform's execution times under varying numbers of accelerator cores and the corresponding speedups. Ideally, the resulting speedup should increase linearly with the number of accelerator cores. For instance, Figure 6.3 shows the relationship between speedup and number of accelerators for the scenario where a single accelerator per FPGA is used, i.e. a maximum of 64 accelerators since the evaluation setup uses 64 FPGAs. Most points of the results plotted are reported in detail through Table 6.2, where the GFLOPS equivalent is also presented. As we can see, a near-perfect scalability has been measured, since the maximum number of accelerators yields an almost identical processing speedup.

Table 6.2: Results when using one Michelsen accelerator per FPGA: Execution time, GFLOPS and speedup

Accelerators	**Exec Time (sec)**	**GFLOPS**	**speedup**
1	27.15	21.6	1.00
2	13.64	42.9	1.99
3	9.11	64.4	2.98
4	6.82	85.9	3.98
6	4.55	129	5.97
8	3.41	172	7.97
12	2.27	258	11.95
16	1.71	343	15.90
20	1.37	429	19.88
24	1.14	515	23.82
30	0.91	643	29.78
32	0.85	686	31.78
40	0.68	858	39.72
48	0.57	1030	47.68
60	0.46	1287	59.56
64	0.43	1372	63.52

A similar pattern has been measured in the case of a maximum of 2 accelerators per FPGA, hence, a maximum of 128 accelerators in total. Nonetheless, a slight slope reduction in the speedup versus number of accelerators plot starts to unravel an important point that needs to be taken into account. Compute-intensive applications such as the Michelsen may rely on considerable data sizes in order to yield results. This can lead to memory bottlenecks that, in turn, begin to pose performance limitations. The Michelsen implementation has been one such case, whereby the accelerator core operates so fast that runs the risk of being starved of input data.

In the case of the Michelsen accelerator, this phenomenon is further accentuated when the number of accelerators per FPGA becomes 3 and 4. Figure 6.4 shows the measured speedup in relation to number of accelerators for parallel execution for all four cases, i.e. 1, 2, 3 and 4 accelerator(s) per FPGA, resulting in a maximum of 64, 128, 192 and 240 respectively. Representative result points are also reported in Table 6.3 along with execution times collected through a number of repetitive measurements. First, it is clear that a one-to-one relation between speedup versus number of accelerators exists only in the case of one core per FPGA. This is almost the case for two cores per FPGA while a distinct slope reduction appears in the cases of three and four cores per FPGA. Naturally, a less data-hungry, in terms of processing speed to data transfer ratio, accelerator core would more easily maintain the one-to-one ratio. Various such revisions of the Michelsen accelerator where deployed to validate this, however achieving linear speedup only in the expense of reduced time to solution, hence, less speedup compared to that shown in Figures 6.3 and 6.4.

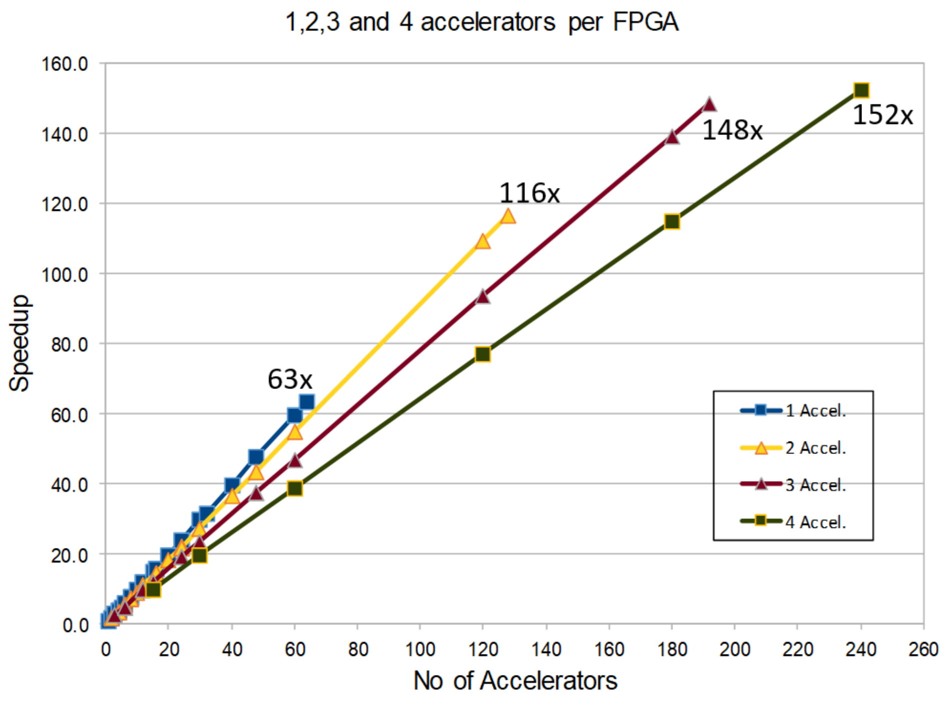


FIGURE 6.4: Speedup vs no. of accelerators, compared to a single accelerator execution, and for varying number of activated accelerators per FPGA

Table 6.3: Representative result points when deploying multiple Michelsen accelerators per FPGA: Execution time (sec) and speedup compared to single-accelerator execution

Total Accelerators	**Accelerators per FPGA**							
	1		**2**		**3**		**4**	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1	27.15	1.00	–	–	–	–	–	–
2	13.64	1.99	14.21	1.91	–	–	–	–
3	9.11	2.98	–	–	10.17	2.67	–	–
4	6.82	3.98	7.37	3.68	–	–	–	–
8	3.41	7.97	3.69	7.36	–	–	–	–
12	2.27	11.95	2.45	11.06	2.75	9.88	–	–
15	1.82	14.93	–	–	2.21	12.29	2.74	9.91
16	1.71	15.90	1.85	14.77	–	–	–	–
24	1.14	23.82	1.24	21.98	1.41	19.25	–	–
30	0.91	29.78	0.98	27.58	1.14	23.72	1.38	19.63
48	0.57	47.68	0.62	43.60	0.72	37.68	–	–
60	0.46	59.56	0.50	54.80	0.58	46.84	0.70	38.90
120	–	–	0.248	109.40	0.291	93.45	0.352	77.2
128	–	–	0.233	116.53	–	–	–	–
180	–	–	–	–	0.195	139.32	0.236	115.1
192	–	–	–	–	0.183	148.45	–	–
240	–	–	–	–	–	–	0.179	152.1

For the particular evaluation setup, the maximum speedup achievable, while also considering accelerator count, can be considered the 148x figure achieved for 192 total accelerator cores in a three-core-per-FPGA setup, offering a favorable consumption of hardware resources to speedup ratio. The 240 core setup improves performance marginally, i.e. raising to 152x, albeit considerably increasing core count. Nevertheless, this is tied to the particular accelerator type. The advantage of UNILOGIC is that it allows for the use of alternative reservoir simulation accelerator types, that could potentially prove to be even more efficient compared to Michelsen. The system's performance is a function of a number of parameters, the most important being accelerator core size and availability of data. The former has to do with accelerator core occupancy of the hardware resources, i.e. the smaller the hardware imprint, the more can fit in a single FPGA. The latter has to do with how quickly the data can be provided to the accelerator core so that it does not idle during processing, i.e. the faster the memory controller throughput or the greater the memory controller count, the faster data can be delivered.

For example, in the case of the Michelsen, the accelerator core processes data very fast, therefore, the data had to be transferred close to the point of processing so that they could be introduced quick enough to the accelerators. Hence, each FPGA had the corresponding data stored in its local DDR while using all four ports of the Processing System in parallel. This way the maximum memory throughput of 9 to 10 GB/s can solely be shared among accelerators of the same FPGA. This way notable performance degradation does not occur in the case of a single accelerator per FPGA, minor deviations have been recorded in the case of two accelerators, while in the cases of three and four, data access bottlenecks begin to play a more considerable role.

6.2 Comparison & Power Efficiency Evaluation

The custom built QFDB, as mentioned in section 4.1, features many power sensors which can provide accurate power consumption measurements for our platform. We measured power consumption of each FPGA that encompasses the complete UNILOGIC architecture, with 4 Michelsen accelerators operating in paraller. This reaches a mere of 8.2 watts, as seen in the 1st row of Table 6.4, including the DDR consumption. As stated before, this hardware configuration yields 73.3 GFLOPS per FPGA, which corresponds to a power efficiency of almost 9 GFLOPS per Watt. This power efficiency is comparable to the Top500's [112] and Green500's [37] GFLOPS/watt measurements [106], which however were measured in 2019 using a 7nm technology and benchmarks that usually perform better than real-life applications [95] [96]. Furthermore, it has the potential to be further improved significantly, as we will explain below in this section.

It is useful to also report that power consumption delta (i.e. the excess power consumed when acceleration is ongoing as opposed to a programmed but idling FPGA), adds up to just 3.6 Watts. On the other hand, if we take take into account the full system consumption, measuring the power entering each chassis whilst acceleration is in full-throttle, and dividing this by the aggregate FPGA count, we end up to an all-included consumption equivalent of 12 Watts per MPSoC device.

TABLE 6.4: Power Efficiency results, both for the quad-Michelsen scenario, as well as for the matrixMult core. Both FPGA power consumption is given, as well as power Delta

Accelerator	**Watts per FPGA (Watt-Delta)**	**GFLOPs**	**GFLOPs per Watt (Delta)**
Michelsen (4x)	8.2 (3.6)	73.3	9 (20.3)
Matrix Mult.	16.2 (11.3)	275	17 (24)

In order to have a power efficiency comparison of our current system acceleration and a high scale alternative, we ported the same application to a recent Dell PowerEdge T630 Server, utilizing all Virtual cores. This Dell server includes 36 cores, or equally 72 virtual cores (threads), based on the Intel Xeon E5-v4 processor family. Compared to a single UNILOGIC chassis (half our prototype), we measured an improvement gained with our approach of 2.67x regarding speedup, and an even more appealing 46x improvement regarding energy to solution, as seen in Table 6.5. Deploying both UNILOGIC chassis almost doubles speedup, now reaching 5.31x, while preserving the same energy efficiency. What is more, the 14 nm technology of the CPU server should be considered at least a generation ahead of our FPGA's 16 nm, which directly affects consumption.

TABLE 6.5: Evaluation of the UNILOGIC-Blade vs. a CPU-Blade

Number of UNILOGIC Blades	**Speedup Over 36-core blade (72 Vcores)**	**Energy Efficiency over CPU-blade**
1	2.67	46
2	5.31	46

Furthermore, in order to compare the UNILOGIC system with GPU execution, we ported the application deploying the Nvidia 980GTX. The execution time for the same dataset reached 3.6 seconds in the GPU, while in Table 6.6 this is compared to a varying number of QFDBs, along with the related speedup and energy consumption measurements. The QFDB related numbers are taken from Table 6.3, and correspond to the worst-scaling scenario of deploying four accelerators per FPGA. The resulting numbers show a single QFDB outperforming the GPU execution by 30%, while being almost eight times more energy efficient. The energy efficiency, reported as energy to solution, remains almost stable as we scale to more QFDBs, while the performance, i.e. time to solution, reaches a ten times improvement, compared to the GPU, when deploying a single UNILOGIC blade, and about 20 times for two blades, i.e. the full prototype.

TABLE 6.6: Comparison on the Michelsen execution on the UNILOGIC platform (15 accelerators per QFDB) versus GPU execution

Platform	**Runtime (s)**	**SpeedUp**	**Energy (J)**	**Energy Efficiency**
GPU (Nvidia 980GTX)	3.63	-	1018	-
1x QFDB	2.74	1.3	131	7.8
4x QFDB	0.70	5.2	131	7.8
8x QFDB (1 blade)	0.35	10.4	133	7.7
16x QFDB (2 blades)	0.18	20.4	134	7.6

Regarding further improvement potential, we should point out that the implemented and primarily used Michelsen accelerator actually has a data transfer to computation ratio that does not favor consumption metrics. However it constitutes our selected accelerator case, as it is better fitted for our prime requirement to stress the system in terms of data traffic. This way we could deeper investigate on the system's unification behavior under stressful traffic. Accelerator cores that apply a higher computation to data transfer ration, benefit even more by the FPGA's low power consumption. We should also mention that we report on our full-prototype design for fairness, which is clocked at 200 MHz, whereas we have single QFDB (4x FPGA) design versions operating at 333 MHz, with respective performance improvement, and comparatively lower power overhead.

Another metric, in a way related to the previous one, is DSP utilization. DSPs are the best performing blocks of the FPGA as far as FLOPs are concerned, and utilized mostly by computation oriented hardware. Our 4x Michelsen accelerator design utilizes just about 30% of the total DSPs available, indicating that the platform is able to reach further improved power efficiency. Accelerator cores that exploit the majority of the DSPs available are certain to reach a much higher efficiency mark.

Coming to confirm this, we report on another accelerator [14] implemented on the QFDB, that reaches an impressive 275 GFLOPs per FPGA. This is likewise measured on a true case scenario, including all the data transfer back-and-forth the off-chip DDR, and was developed by the FORTH team[1] under the ExaNeSt

[1]Foundation for Research and Technology - Hellas (FORTH), www.forth.gr

project [62]. This accelerator corresponds to a highly optimized and MPSoC-tailored matrix multiplication core. This core is parameterizable to different sizes, with the reported performance achieved for a version utilizing 82% of DSPs, and 56% of LUTs. The relatively low LUT utilization allows this design to fit in the UNILOGIC framework, as the supporting excess logic requires just about 12% to 23% of LUTs. As for power efficiency, and looking in the 2nd row of Table 6.4, again power sensor monitoring, cross verified with lab tests observing lab power supplies, yielded 16.2 Watts for this accelerator version, which accounts for an almost doubled efficiency of 17 GFLOPs per watt.

Finally it's worth reporting that our selected Worker device is a medium sized MPSoC, considering the current MPSoC range. As an example, even if we consider the currently available XCZU15EG MPSoC, which is pin-compatible to our XCZU9EG, it includes 25% more LUTs and 40% more DSPs, and is thus capable to offer a proportionate improvement in term of FLOPs at a very modest consumption increase (as only the extra logic/DSPs' consumption is to be accrued). Furthermore, based on currently available technologies and the graceful scaling characteristics of the UNILOGIC architecture, a projection to exascale performance can be found in Appendix E.

6.3 Case 2: Evaluation With the Compute-Intensive KNP Accelerator Core

In order to provide additional results on the performance of UNILOGIC, a second accelerator type has been introduced to the evaluation platform, namely the KNP kernel. This kernel is based on Lucas Kanade's Optical Flow algorithm and it is widely used in monitoring average road traffic speed. This is achieved by using video streams coming from standard surveillance road cameras, as seen in Figure 6.5, while more details about the kernel implementation can be found in [5].

For the KNP kernel, the evaluation setup differs to that of the Michelsen in that the FPGA can host a maximum of two accelerator modules. Hence, comparatively, KNP takes up more hardware resources than the Michelsen. On the other hand, it processes data at a rate that allows for remote storage. This means, that under

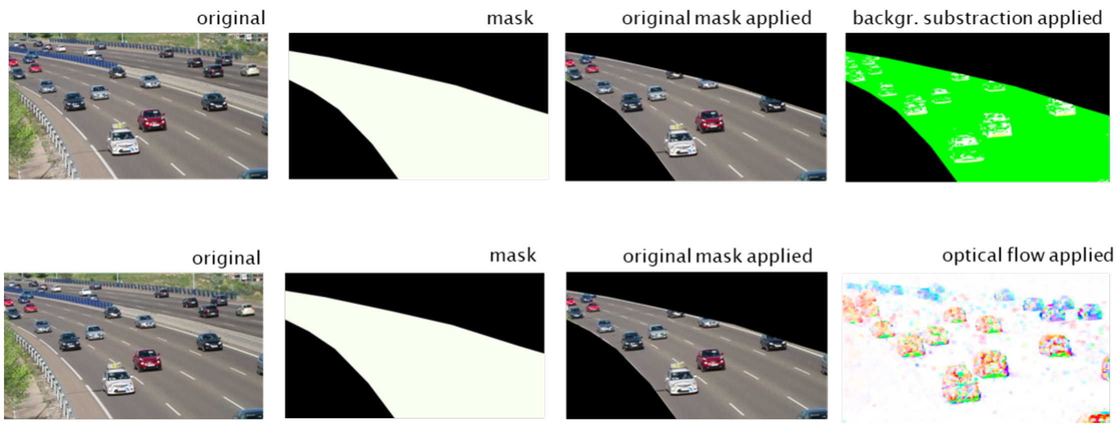


FIGURE 6.5: Optical Flow algorithm presentation - KNP

the UNILOGIC optimizations for cross-FPGA acceleration, data do not need to be stored locally for avoidance of delays incurred due to lack of accelerator input data, as could be the case for Michelsen. Instead, it is the memory of a single FPGA in the QFDB that is solely used for data storage, and thus all the accelerators efficiently access the same memory, mostly benefiting from the UNILOGIC architecture. So, as the KNP accelerator is compute intensive, the UNILOGIC implementation helps to avoid memory contention. Thus the execution time is not affected significantly by just using a single memory data source. This helps illustrate the functional advantages of UNILOGIC; a single storage point for the input frames is needed for all the accelerators that work on different parts of it, while no performance degradation is incurred although remote DDR is used.

The execution was initially performed using one QFDB, which, as shown in Table 6.7, achieves the real-time requirements for such an application. Subsequently, a second QFDB was deployed so that workload is shared between two QFDBs in order to provide a significant runtime margin as I/O from the platform to a remote server or workstation may add additional overheads. Similarly distinct, real time streams can be processed in parallel. Each QFDB can be assigned to process different video streams from different cameras in real-time.

The evaluation results in Table 6.7 as well as Figure 6.6, clearly show that the scalability of the specific application running on multiple QFDBs is guaranteed. That is, the performance increase is linear to the amount of accelerators deployed across all accelerators from both QFDBs. Finally, Table 6.8 provides additional information on how the execution of the KNP kernel on different processing platforms compares. These are a CPU, a GPU, two FPGA architectures and, finally,

TABLE 6.7: Execution times of the KNP kernel in up to 2 QFDBs (8 FPGAs) for 60 frames

Accelerators per FPGA				Execution	Number of	Relative
F1	**F2**	**F3**	**F4**	**time (s)**	**accelerators**	**speedup**
1				6.40	1	1.00
2				3.20	2	2.00
2	1			2.25	3	2.85
2	2			1.63	4	3.92
2	2	2		1.11	6	5.78
2	2	2	2	0.81	8	7.89
3	3	3	3	0.55	12	11.64
4	4	4	4	0.40	16	15.81

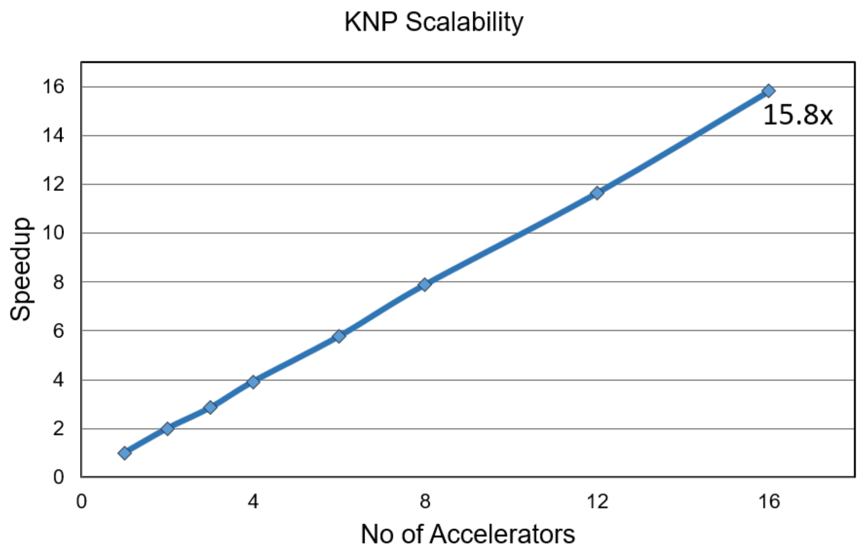


FIGURE 6.6: Optical Flow algorithm (KNP) speedup while activating multiple accelerators, compared to a single accelerator execution

the UNILOGIC scheme with a pair of QFDBs. The listed results suggest that the density provided by the QFDB, in conjunction with UNILOGIC, create a processing setup that can achieve significant results with respect to performance, especially for applications suited for parallel execution such as KNP.

Specifically, utilizing a single QFDB outperforms the CPU implementation by a factor of 446 and the GPU by a factor of 3. Moreover, it outperforms the other FPGA implementations by a factor of 2.7, mostly due to the extra parallelization provided by the utilization of a QFDB's 4 FPGAs. Also, employing a second QFDB, linearly increases the performance gains over the other platforms by a factor of 2. Slight slope variations occur mainly due to imperfect dataset distribution.

TABLE 6.8: Performance and Energy Efficiency per frame for the KNP algorithm

Parameters	**CPU Intel Xeon @3.5GHz**	**GPU NVIDIA GeForce GTX960**	**FPGA**			
			Virtex 7	**Virtex Ultrascale+ (AWS-EC2)**	**1x QFDB (8 accel.)**	**2x QFDBs (16 accel.)**
Device time (ms)	5925.8	42.7	36.3	37.3	13.3	6.7
SpeedUp vs. CPU	1	139	163	159	446	884
Device power (W)	10	75	8.4	8	12	24
Energy (mJ)	59258	3202.5	304.9	298.4	159.6	160.8
Energy Efficiency	1.0	18.5	194.3	198.6	371.3	368.5

With respect to energy efficiency, the results are also shown in Table 6.8. The power consumption measurements of the CPU and the Ultrascale+ FPGA are taken from the power reporting tool of the Amazon AWS, while the GPU power consumption was measured using the NVIDIA System Management Interface (NVIDIA-SMI).The power consumption of the Virtex 7 FPGA was estimated using the Xilinx Power Estimator (XPE). Finally, the QFDB's power consumption was measured using the dedicated power measuring circuitry provided by the board. The power consumption presented refers to the power consumption of each respective platform minus the power consumption in the idle state. From these results we can see that the FPGAs offer an impressive two orders of magnitude less energy consumption than a CPU and one order of magnitude less than a GPU. Finally, all the FPGA implementations offer comparable energy efficiency with the QFDB being almost twice more efficient.

Chapter 7

Future Work

This section addresses the main challenges and issues that are currently under investigation that will result in a complete and robust framework that has UNILOGIC at its core. Hence, we are currently working on extending the runtime system with new features which will significantly increase the productivity of developers. This includes the development of a toolset for a fully-automated development flow of UNILOGIC-compatible hardware accelerators, as well as a toolset for the decentralized management of a global acceleration library. Building such an acceleration library is also an important part of the ongoing work, with several widely-used math modules evaluated so that they can be added. Furthermore, we are experimenting with efficient HPC application candidates in other areas such as potential well analysis which is key in quantum mechanics, shock polars being important in aerodynamics, plane frame analysis used on structural as well as civil engineering, complex chemical equilibrium and satellite space imaging [92, 57].

Moreover, we investigate emerging, newly launched solutions, which may well allow for the implementation of the UNILOGIC architecture in an even more efficient manner. Such candidates include the newly introduced Xilinx Versal Adaptive Compute Acceleration Platform (ACAP), which combines processors and reconfigurable logic rich in DSPs, together with Vector processors; it also deploys transceivers that bring off-chip throughput to the Tera-bit/sec range. As another candidate, the recently launched Xilinx Alveo Data Center accelerator cards feature reconfigurable solutions that overcome both (i) memory saturation points, through the use of High Bandwidth Memory (HBM2) offering 460GB/s bandwidth, as well as (ii) interconnection bottlenecks with a few Tera-bits/sec of aggregate transceiver throughput. In addition, Intel, which formerly acquired Altera, encompasses the GX/GXT/GXE transceivers, reaching up to 57.8 Gbps.

Intel has also delivered the 10nm Agilex FPGA SoC family [19], also incorporating ARM A53 processor cores and featuring transceivers capable of up to 112 Gbps, increased DSP capabilities and coherent interconnect to Intel Xeon processors, providing a platform for high performance and low power solutions, for data center, networking, as well as edge computing applications. Based on the UNILOGIC characteristics, i.e the exploitation of powerful FPGA-to-FPGA interconnection as well as high memory bandwidth, it is expected that those new devices will allow for even more efficient implementations of HPC systems based on the proposed architecture.

Chapter 8

Conclusion

One of the most active research areas in computer architecture is that of the HPC systems. An important challenge is the energy efficiency of today's HPC systems, which cannot be improved simply by scaling the numbers of CPU cores alone. This motivated a move towards heterogeneous systems including GPUs, Vector Processors and FPGAs.

This thesis presents the UNILOGIC architecture for deploying and programming HPC heterogeneous systems, incorporating multiple CPUs and FPGAs. In order to increase programmability, the architecture offers a unified environment where all the reconfigurable resources can be seamlessly used by any processor/-operating system. Moreover, the architecture provides hardware virtualization of the reconfigurable logic so that the hardware accelerators can be shared by several applications or tasks.

In terms of performance, we show that the architecture can efficiently scale while it introduces a low overhead in the communication latency. In particular, to evaluate UNILOGIC, we have built a prototype consisting of 64 Xilinx MPSoCs each incorporating 4 ARM Cores and significant reconfigurable logic. Based on the evaluation of two real-world data and compute intensive applications, UNILOGIC scales almost linearly while it allows for all the reconfigurable resources in the parallel system to be utilized as if they were in a single large device. The performance overhead in order to support remote accesses of reconfigurable resources is less than 4%.

Finally, our evaluation demonstrates that the energy efficiency offered by the prototype is comparable to state-of-the-art HPC systems, using newer transistor technologies. In particular, it offers 9 to 17 GFLOPS per watt using the Xilinx 16nm UltraScale+ devices.

Appendix A

OpenCL and Virtualization

In this Appendix we try to give a deeper insight on how the Open Computing Language (openCL) kernel calls function. We are using some explanatory depictions through a number of images, and then we proceed to the Virtualization addition. Starting with Figure A.1, we see how a traditional, openCL version 1 (i.e. versions 1.0, 1.1, 1.2) operates. As there was yet no shared memory support, in a heterogeneous system that e.g. includes an FPGA and a host CPU machine, a lot of data transfer needs to take place. Looking at Figure A.1, we see that the data, labeled "D", reside in the host memory, while the accelerator needs to access them. So data are 1) transferred to the FPGA's DDR, then 2) the accelerator transfers the data inside the FPGA, i.e. into Block-RAM so that as a next step, X) the actual eXecution, i.e. the hardware computation takes place. Then 3) results are transferred to the FPGA's DDR, and at a final step 4) results are transferred back the host's system memory (designated as Global Memory in openCL terminology).

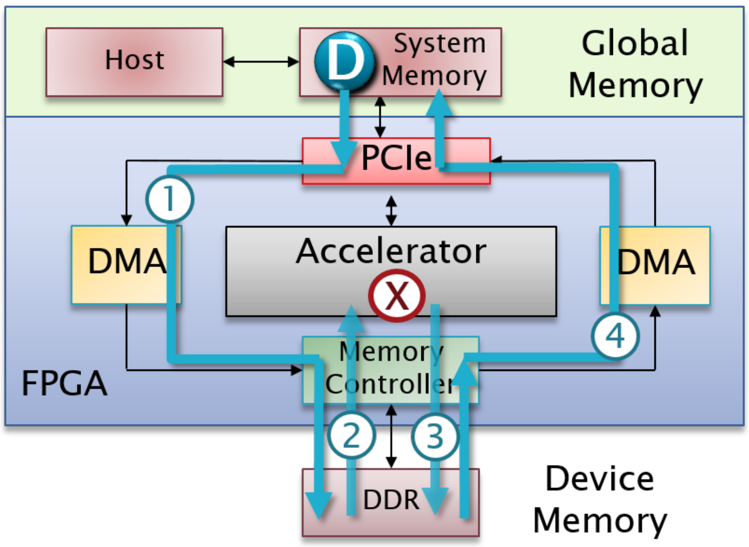


Figure A.1: Traditional OpenCL Data Movement

An important improvement came with openCL version 2 (i.e. versions 2.0, 2.1, and 2.2 being the current version), as it included shared memory support. Under this update, the FPGA+HostCPU approach now becomes much more efficient, as can be seen in Figure A.2. The FPGA/accelerator now has access to common, shared memory, so no data relocation is needed. Now data are 1) requested directly by the accelerator, which then X) eXecutes the corresponding computation, and then 2) sends the results directly back to the shared memory.

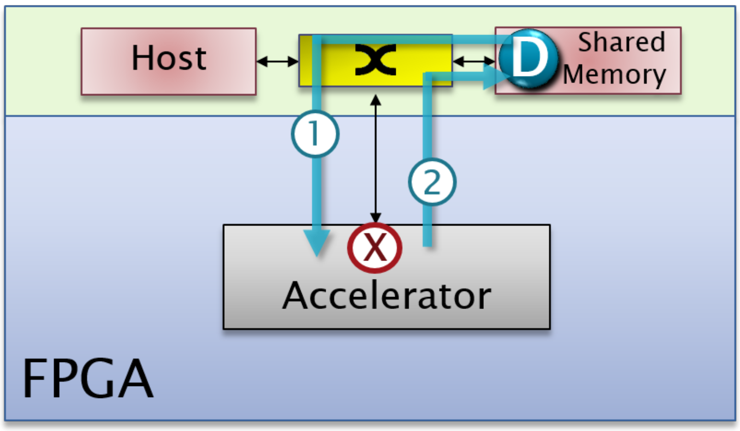


FIGURE A.2: OpenCL 2.0: Shared Memory

When more than a single Worker, i.e. FPGA+HostCPU bundle, is deployed, a suitably configured local and global interconnect can allow for remote accesses, as can be seen in Figure A.3. In this example, a kernel call, labeled "C", 1) invokes an accelerator that resides in a remote FPGA. So Worker 0 asks to be served by Worker 1. What is more, it instruments the remote accelerator to use the data "D" that reside in Worker 0. So the designated accelerator X) proceeds with the execution step "X", which need to access and operate on remote data. However, just as Figure A.2, no extra data migration is needed, but rather the remote accelerator directly accesses remote data.

In order to have acceleration sharing, a virtualization mechanism needs to be encompassed in the architecture, and this can be seen in Figure A.4. Now anyone who needs to invoke an accelerators does not directly access it, but rather it accessed the Virtualization block which enqueues the call description, and then properly dispatches it to the accelerator. The virtualization block can be accessed by from any Worker, as it can be addressed and reached through the local/global interconnect. In the example of Figure A.4, both Workers 0 and 1 execute a

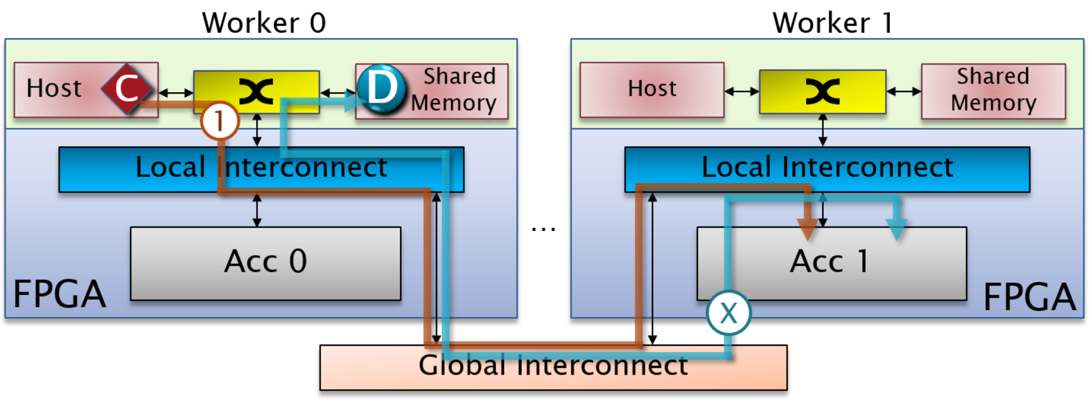


FIGURE A.3: Multiple FPGAs: Remote Invocation and Remote Memory

different call "C", that however needs to invoke the same accelerator, which reside in Worker 2. The accelerator itself need not be accessed, but rather both 1) the call from worker 0 and 2) the call from worker 1, ask to be serviced by the Virtualization block in Worker 2. That block enqueues both calls, and will dispatch them when the accelerator is available. The data in turn are stored locally to the originating Workers, so the accelerator "Acc 2" will need to access them remotely. If the blue call by Worker 0 is dispatched first to the accelerator, it executes (X1) accessing data from Worker 0. Then, the Virtualization block dispatches the second call to the same accelerator, and so it gets executed (X2), accessing remote data in Worker 1.

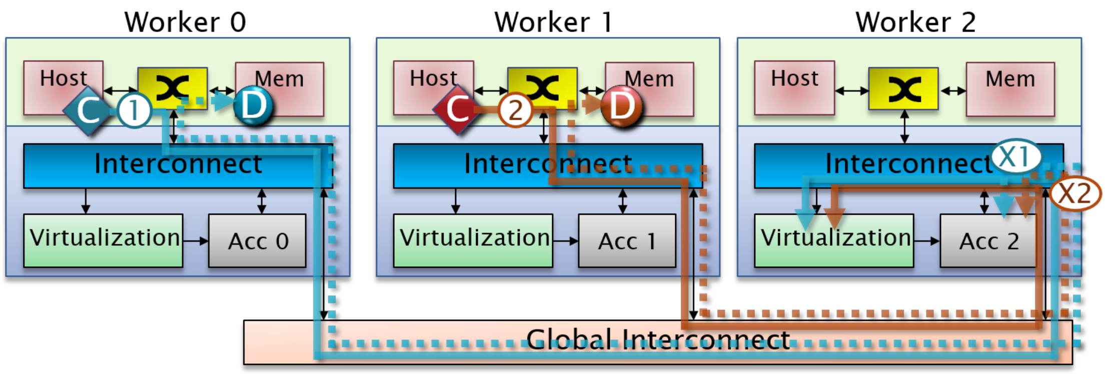


FIGURE A.4: Multiple FPGAs: Resource Sharing

Finally, in Figure A.5, we see how the virtualization scheduler can also achieve fine-grain sharing of the accelerator resources. This discussion also corresponds to the analysis we have seen in section 3.1, and on Figure 3.3 which for ease of access is also copied here as Figure A.6. In our example of Figure A.5, three kernel calls

"C" from Worker 0, and two more from Worker 1, need to employ the accelerator in Worker 0. Hence, they both access the corresponding Virtualization block, and the call descriptions are enqueued therein.

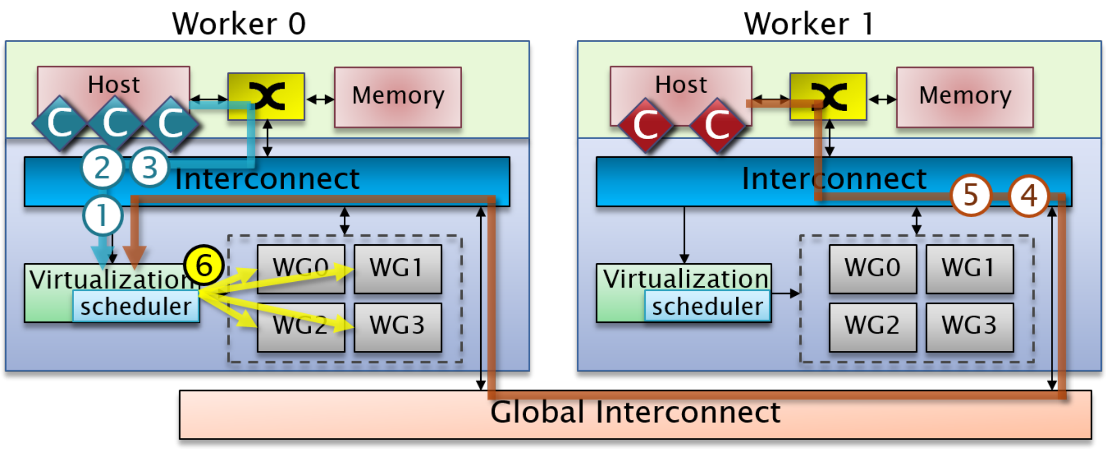


FIGURE A.5: Multiple FPGAs: Fine-grain Resource Sharing

Importantly, this block can control many hardware copies that implement the same Work Group (WG) description, i.e. many identical accelerators, and orthogonally, it can divide a big kernel call into many WG calls that directly map to the hardware WGs attached, as also seen in Figure A.6. The resulting WGs are then dispatched accordingly to the hardware WG modules, labeled as step "6" in Figure A.5. This may result to local or remote data accesses, as was seen in the previous examples, however not depicted here for simplicity.

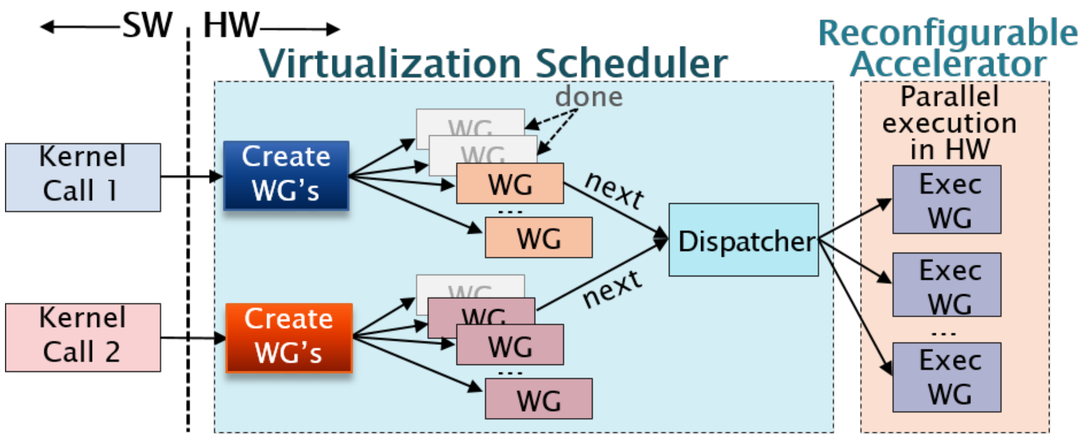


FIGURE A.6: OpenCL Kernel WG Distribution with the Virtualization Scheduler

Appendix B

Worker (FPGA) Block Diagrams

B.1 F1 Block Diagram, Excluding Reconfiguration

The block diagram of Figure B.1 is exported from the Xilinx Vivado tool, and corresponds to the F1 FPGA of the QFDB. As this is the "Network" FPGA, it encompasses more networking elements, in order to realize the final prototype's topology. This can be seen on the upper half of the page, with 8 outgoing paths, driven by 3 double and 2 single C2C modules. These are standing in between forward and backward address translation blocks, while also augmented by convID blocks intervening to the outgoing path, i.e. just before the slave side of the C2C. On the top of the page, we can see the hierarchical central AXI interconnect, that corresponds to Figure 4.36, and offers the –selective– all-to-all connectivity among accelerator controllers, memory ports and network interfaces (C2C blocks), in order to unify local as well as remote resources. In the middle stands the Processing System, and the related address translation blocks, separated for accesses to memory and to peripherals. Then in the bottom, we see on the left the blocks for the final translation to access local memory, pertaining to the discussion on Figure 4.33 of section 4.2.5. On the right side we see the accelerators highlighted in orange, driven by the accelerator controllers, i.e. the mailbox/scheduler pairs. Following those, and intervening on the outgoing paths, i.e. to the transactions initiated by the accelerators, come the forward address translation blocks.

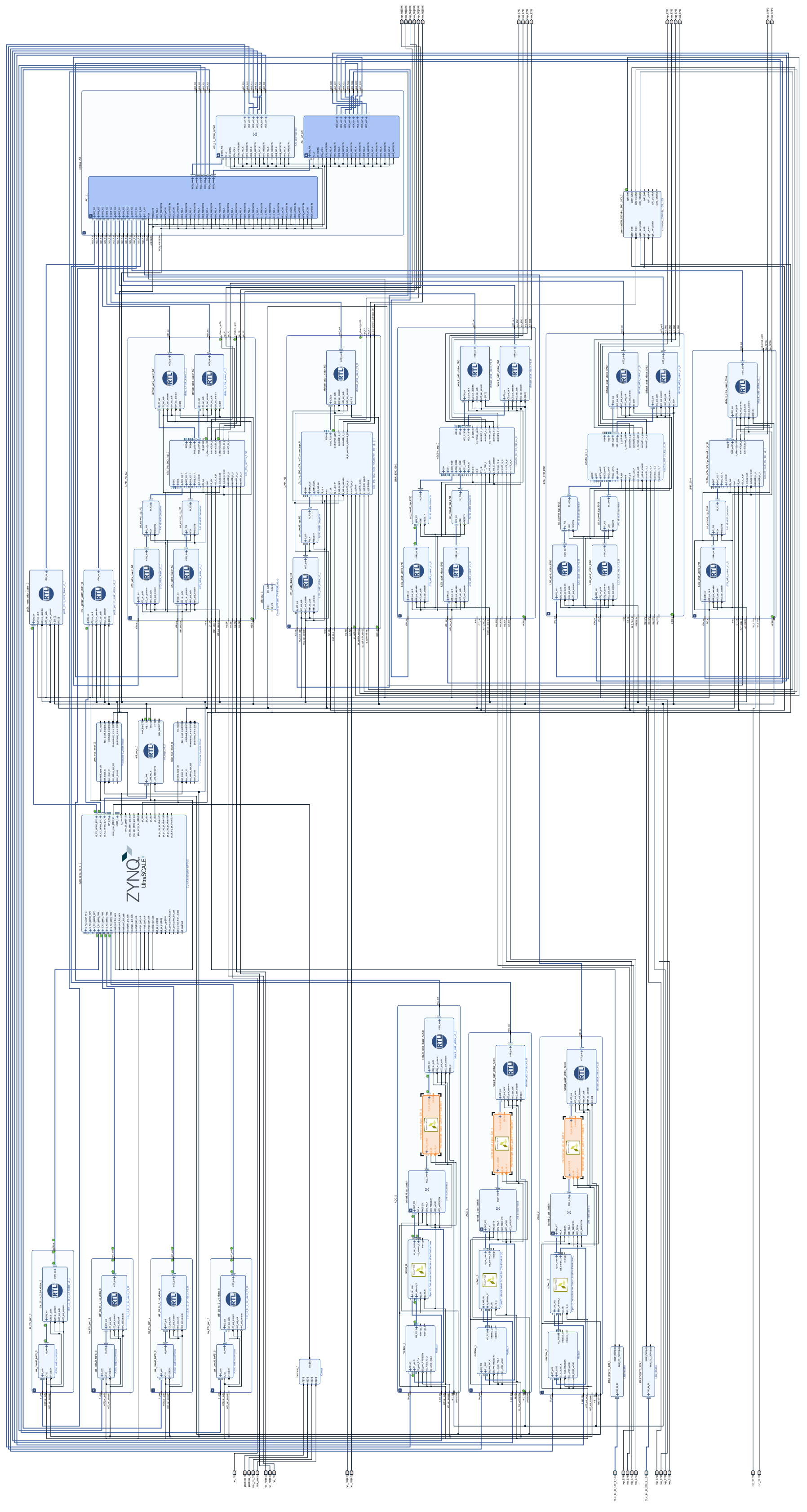


FIGURE B.1: Network FPGA (F1) block diagram, including accelerators & UNILOGIC, while excluding reconfiguration

B.2 F3 Block Diagram, Deploying Reconfiguration

In order to display a block diagram that incorporates the partial reconfiguration logic, we have chosen the F3 FPGA, which, just like the F2 and F4 ones, includes less networking logic, hence it looks less complicated, making it easier to focus on the reconfiguration infrastructure. In Figure B.3 we see the F3 block diagram, as exported by the Xilinx Vivado tool. The accelerators, which constitute the focal point for these designs, are seen close to the center, and are highlighted in orange. These are now actually substituted with Partial Reconfiguration (PR) Slots, and "Above" each such slot, stands the associated decoupler block. On both sides, i.e. master and slave interfaces, of each slot also stand discrete AXI register slices blocks, in order to completely isolate the partially reconfigurable region from the static one, and thus allow higher clock frequencies to be achieved. In Figure B.2 we have highlighted a single such slot and the accompanying blocks, and should mention that this part of the design pertains to the investigation presented in section 4.3 and involving Figure 4.41. Around this infrastructure, logic also exists, that accommodates the more complicated clocking and reset distribution, required to properly drive the 'dynamic partial reconfiguration' related blocks. Just as we have seen in the previous Figure B.1 for F1, before the accelerators and at the bottom of the page, comes the accelerator controllers, and at the top of the page we find the much simpler networking infrastructure, along with the blue-highlighted central AXI interconnect. Just above the center, stands the Processing System along with associated address translation blocks.

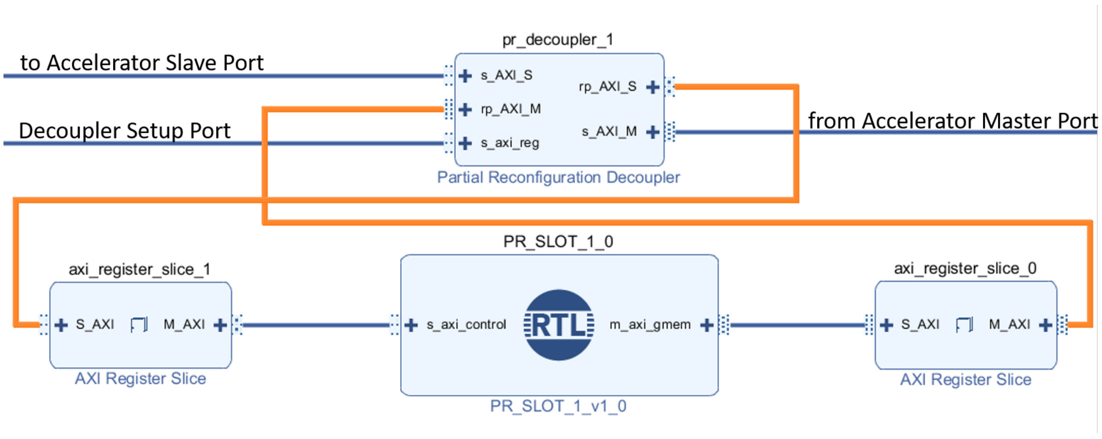


Figure B.2: Highlighting the Partial Reconfiguration (PR) slot, along with blocks providing proper isolation

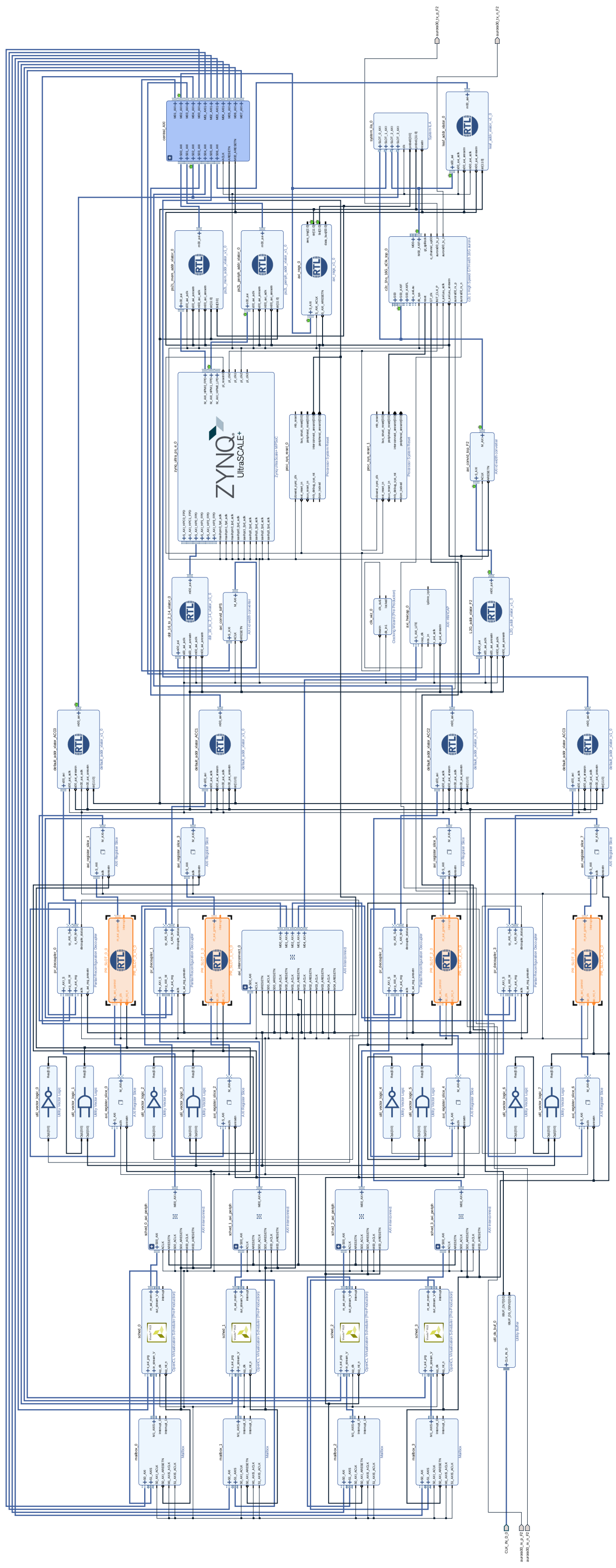
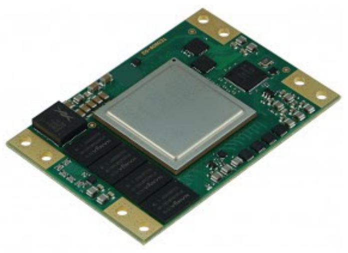


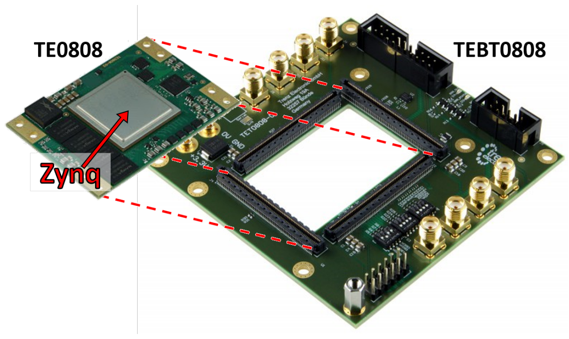
Figure B.3: F3 FPGA block diagram, including UNILOGIC & accelerators along with reconfiguration

Appendix C

Initial MPSoC Exploration & Deployment

As we have already stated in section 5.1.1, before moving on to single- and multi-FPGA optimizations for the UNILOGIC architecture, a lot of research and development effort has been invested, regarding the newly launched –at that time– Xilinx Zynq Ultrascale+ devices, comprising a Multi-Processor System on Chip (MPSoC). These devices where the first to combine the Arm v8-based Cortex-A53 high-performance energy-efficient 64-bit application processor and the UltraScale architecture to create the industry's first programmable MPSoCs. The focus was on low total power consumption, heterogeneous processing, and programmable acceleration. One of the very first available devices by Xilinx was the xczu9eg, within the package ffvc900-1-es. The first commercial board available, hosting this device, was provided by Trenz, labeled "TE0808" and can be seen in Figure C.1a.


(A) TE0808: The first mini board by Trenz, hosting a Xilinx Zynq Ultrascale+



(B) TEBT0808: A test board provided by Trenz, for testing the TE0808 Zynq module

FIGURE C.1: The first available commercial module by Trenz Electronic to host a Xilinx Zynq Ultrascale+ FPGA

We had our hands very early on such devices, and immediately started to investigate on these, in order to examine its suitability for the purposes of the UNIMEM and the UNILOGIC architectures. At that time this was of great interest both for the ECOSCALE project incorporating the UNILOGIC approach, as well as the ExaNeSt and ExaNoDe sister projects. In order to expedite our research, we were supplied by Trenz a first test board to host the TE0808, and allow for power as well as external connectivity. This board was labeled "TEBT0808" and can be seen in Figure C.1b. The TE0808 System on Module (SoM) included 2 GB of 64-bit wide DDR memory, programmable clock generators and QSPI flash memory. The TEBT0808 test kit additionally provided SMA connectors for both PL and PS connected multi-gigabit transceivers, power supply and combined UART/JTAG access through the supplied "TE0790" adapter. All the initial exploration and testing on this first MPSoC was also of great value for the succeeding QFDB design by FORTH. One primal aim of our exploration was to analyze the suitability of the Zynq devices for the QFDB purposes, and in the next steps helped a lot in the specialized QFDB's PCB design.

Our investigation included many aspects of the newly introduced device. One of the main concerns was to study the Processing System (PS) operation, components and addressing, as well as the way it interfaces to the programming logic (PL), i.e. the FPGA part of the device. It was also accompanied by newly launched versions of the Xilinx Vivado and SDK tools that included features to support the Zynq MPSoCs. We moved on to configuring the processor, included baremetal tests to access various aspects such as UART connectivity, QSPI programmability, clock generator configuration and DDR memory testing. We then carried on with configuring the FPGA part of the device, that included the first transceiver Bit Error Rate Testing (BERT), and eventually custom blocks to verify and deploy the interoperability of the PS and PL parts of the MPSoC. This led to a first mini platform, presented in Figure C.2, that included two MPSoCs and an intermediate switch. The switch was deployed in a Xilinx Virtex 7 FPGA hosted on the Hitech-Global's HTG-700 development board.

In order to bring this first prototype to a meaningful functional state, included a lot of added exploration. We had to advance from previous FPGA platforms, that usually included FPGAs, hosting microblaze soft processor cores as processing elements. One other improvement came with the GTH multi-gigabit transceivers,

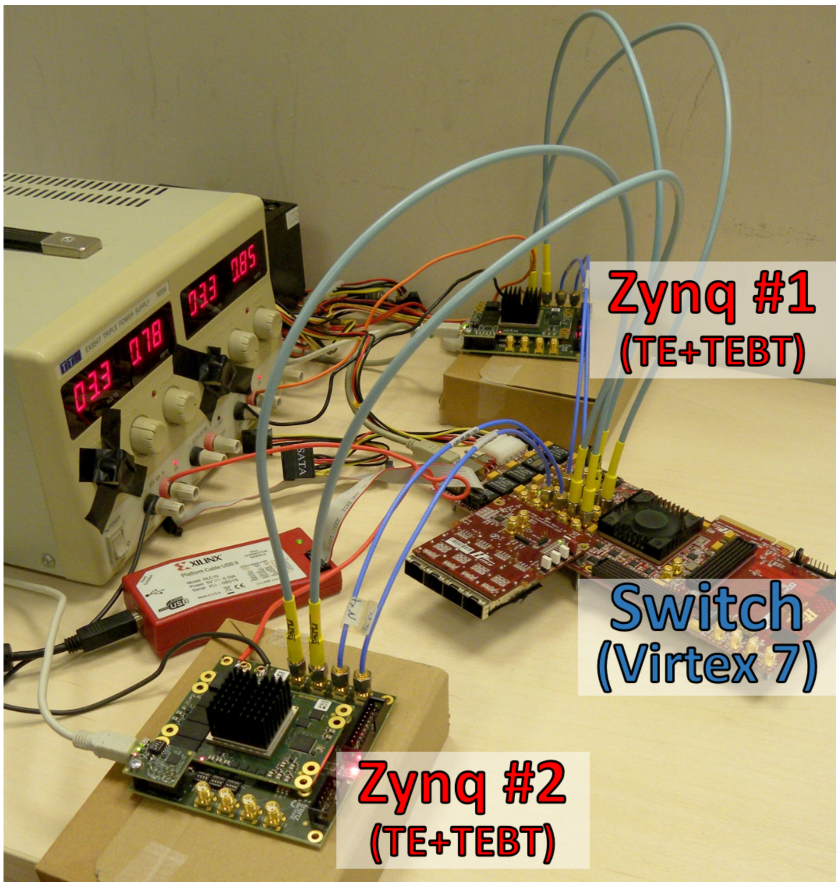


Figure C.2: A lab setup with two TE0808, interconnected through an custom FPGA-hosted switch

now reaching up to 16.3 Gbps. We have initially used GTH transceivers during a first porting of the UNIMEM architecture in a Xilinx KCU 105 board incorporating a Kintex Ultrascale FPGA. This upgrade to GTH transceivers needed a lot of changes to the Chip-to-chip (C2C) module that encompassed the transceivers for cross-FPGA communication, as well as proper deployment of the newly introduced dedicated transceiver clocking modules. A thorough feasibility study for the MPSoC I/O and related FPGA designs was composed at this point, to help the choices on the foreseen custom platforms to be build. Also, the aurora module encompassed in the C2C was now offered with a partly altered interface, and required corresponding upgrades to the surrounding logic. Just as a short reference on our transceiver investigation, prior Xilinx FPGA devices incorporated transceiver flavors, such as the GTP with 6.6 Gbps maximum throughput, the GTX as the ones used in the switch of Figure C.2, reaching up to a maximum of 12.5 Gbps (for -3 speed grade parts), and the GTZ reaching up to 28 Gbps and deployed only in a limited range of expensive devices.

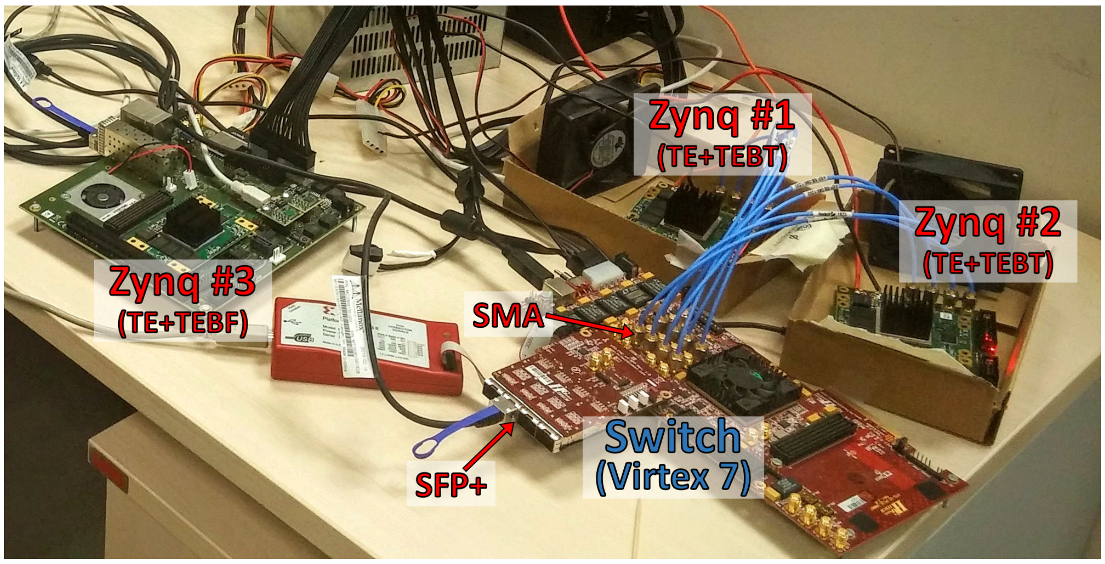


Figure C.3: A lab setup with two TEBT (early boards) and one TEBF (succeeding boards), interconnected through a custom FPGA-hosted switch

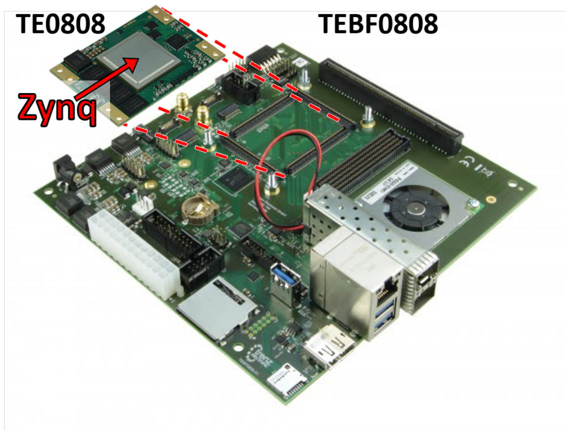
The UNIMEM architecture was then ported to this new Zynq MPSoC platform, with many changes needed due to the different nature of the integrated processing system. New baremetal software was programmed through the latest SDK tool version, tailored to the needs of the Processing System 's build-in address space. The new FPGA's clock managers had to be properly deployed. Proper care was further needed to correctly program both the PS and the PL of the combined system, and then automate this process, either through tool scripts, or by SD-card and QSPI combined sw/hw (boot.bin) programming. Eventually a fully functional UNIMEM infrastructure was set-up, and subsequently accelerators along with basic parts of the UNILOGIC modules such as virtualization were encompassed, in parallel with the initial exploration on the PS address space utilization. Remote DMA transfers were also preliminary evaluated at that time, which in parallel helped stress cross-FPGA connectivity.

Eventually, the latest carrier board by Trenz, the TEBF0808, was launched, and was soon deployed to our exploration platform. This board provides a whole range of on-board components to test and evaluate the Zynq Ultrascale+, and offers SFP+ cages for easier cabling than that of the SMAs used by the previous boards, as can be seen in Figure C.3. A lot of transceiver/link BERT testing was performed, in order to see the potential capability for the inter-FPGA connectivity. A great deal of the constructed link-testing designs were passed to and very frequently used throughout the whole evolution of our research, and are still in effect for the existing platforms deploying multiple FPGAs. Also, at the initial stages, we used those tests to investigate on passive copper and active optical SFP+ cables, and we came to the conclusion that the copper cables where had no disadvantages while in parallel being easier to use and much cheaper to acquire. Finally, all these efforts proved beneficial for the designs to be later deployed on the QFDB-based platforms, and also greatly helped the QFDB bring-up process itself.

Appendix D

Earlier UNILOGIC Investigation on both Michelsen & Hyperbolic Cores

In our initial exploration on the UNILOGIC architecture, we focused on two algorithms, the Hyperbolic and the Michelsen. And both where co-designed within the UNILOGIC implementation on our earlier Trenz-based platform, and got optimize and evaluated. These both pertain to a method that aims at maximising the hydrocarbon recovery of an oil field. i.e. Reservoir Simulation (RS).



(A) TEBF0808: The latest Mini-ITX carrier board for hosting the TE0808 Zynq module

(B) The latest TEBF/TE 0808 enclosure by Trenz, including power supply and cooling

Figure D.1: The prototype used in our initial experiments on the UNILOGIC architecture

RS is the state-of-the-art technology for predicting field performance. This tank reservoir model has been analysed by employing, at different grid points, the Rachford-Rice equation. This is an important equation since it provides insight as to the liquid and vapour elements within the porous material of the rock substrate, and as we have already mentioned in section 5, the algorithm that solves for the

Rachford-Rice equation is the Newton-Raphson method. This is a complicated method that involves multiple iterations at each grid point in order to determine the composition of liquid and vapour elements at each timestep of the simulation. Hence we moved on with the implementation of the two aforementioned kernels that are variations of the Newton-Raphson method, specifically tuned for the RS problem. The respective OpenCL kernels used are Hyperbolic and Michelsen and provide a faster convergence in solving the Rachford-Rice equation for oil reservoir simulation.

Table D.1: Execution time (ms) for 100K elements running on a number of simple (i.e. only basically optimized) accelerator cores, for both Michelsen and Hyperbolic algorithms, both running on a 166.67 MHz clock.

Num of simple cores	**Michelsen ms & (speedUp)**	**Hyperbolic ms & (speedup)**
1	875 (–)	931 (–)
2	438 (1.99)	468 (1.99)
3	295 (2.96)	313 (2.97)
4	222 (3.94)	238 (3.91)
5	177 (4.94)	204 (4.56)
6	149 (5.87)	185 (5.03)
7	134 (6.53)	166 (5.60)
8	117 (7.48)	151 (6.17)
9	–	129 (7.21)
10	–	117 (7.96)
11	–	108 (8.62)
12	–	104 (8.95)

At this first step, the generated accelerator cores have been evaluated on the UNILOGIC architecture, using an intermediate prototype, that is comprised of two Workers. However, at this point no remote acceleration was tested, and the cores have been mapped to the reconfigurable fabric of one of the Workers, that can be seen in Figure D.1. Measurements have been obtained with respect to the execution time of the two algorithms on two data sets of different size, i.e. data for 100K and 200K grid points. Subsequently, the measured times have been compared against those for the identical procedure but on a different execution platform, i.e. i5 Quad-core CPU at 3.1 GHz.

Based on our optimization process, both manually as well as through the specialised Design Space Exploration (DSE) tool provided by Polytecnico di Torino

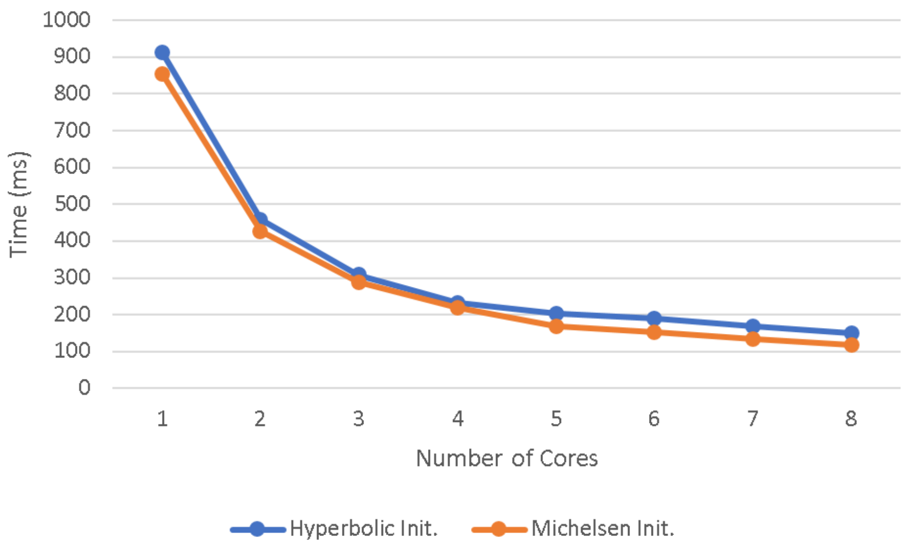


FIGURE D.2: Simple Core Execution time (ms) vs. No. of Cores

TABLE D.2: Execution time (ms) for 100K elements running on a number of optimized accelerator cores, for both Michelsen and Hyperbolic algorithms, both running on a 166.67 MHz clock.

Num of opt/zed cores	**Michelsen ms & (speedUp)**	**Hyperbolic ms & (speedup)**
1	28.5 (–)	26.1 (–)
2	15.3 (1.86)	13.3 (1.97)
3	11.9 (2.40)	8.9 (2.93)
4	9.4 (3.03)	6.7 (3.90)

(Polytechnic University of Turin), we ended up with two versions of each algorithm to be tested. The minimally optimized -simple- versions, which are also occupying minimum FPGA resources, and the optimized versions. We were able to fit up to eight simple Michelsen accelerators in a single FPGA to operate in parallel, whereas we could fit twelve of the Hyperbolic accelerators. In the optimized versions, we were able to fit four in each case, i.e. the same for Michelsen and for Hyperbolic. The results can be seen in Tables D.1 and D.2, while the best results can also be seen in D.4.

Also looking at the plotted results, in Figure D.2 we see the 100K element plot for up to eight hardware accelerators running in parallel. These are controlled by a single Accelerator Controller, i.e. a Virtualization Scheduler-Mailbox pair. We can see at that at this version, the acceleration time to solution scales well to the number of accelerator cores. It even continues to scale well for the twelve Hyperbolic cores, not included in the plotted that compares the two cores. In

TABLE D.3: Best software execution times (ms) form single- and four-threaded CPU execution

	CPU 1 th.		CPU 4 th. (OpenMP)	
Data size	100K	200K	100K	200K
Hyperbolic	25.5	50	8.8	17.2
Michelsen	29	56	11.7	22.3

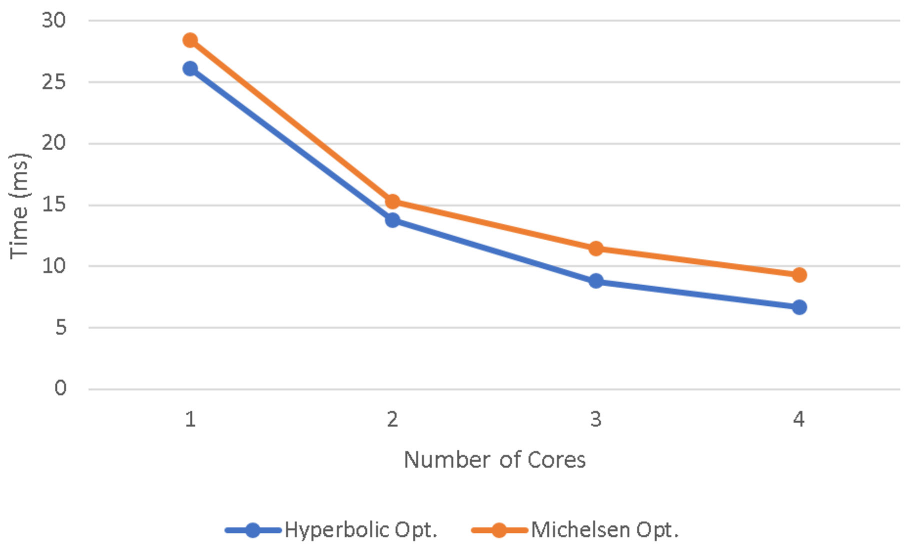


FIGURE D.3: Optimized Core Execution time (ms) vs. No. of Cores

Figure D.3 the optimized core performance is plotted. The execution time is now impressively improved, being an order of magnitude lower. Both scale well even at this performance level, with the Hyperbolic being slightly better.

TABLE D.4: Best hardware execution times (ms) for simple and optimized FPGA accelerators

	Initial 8 Core		Opt. 4 Core	
Data size	100K	200K	100K	200K
Hyperbolic	151	298	6.7	13.8
Michelsen	117	235	9.3	18

Moving on, the execution times achieved by the optimised cores have been compared against those achieved by using conventional processing methods, i.e. a CPU with either a single-thread or a four-thread execution using OpenMP. The best times achieved using these two SW approaches can be seen in Table D.3, and was also presented in section 5.1.1.2. Those achieved by the two sets of the implemented hardware cores, i.e. simple and optimised, and on two different data sets have been gathered in Table D.4. It is clear that the execution of the optimised accelerator cores on reconfigurable hardware yields the best performance and this is better displayed in Table D.5. First, the optimised cores offer a speedup of 3.8 and 3.1 over single-thread CPU execution for the Hyperbolic and Michelsen algorithms respectively. Even with a four-thread CPU execution a speedup is noted of 1.3 and 1.2 for Hyperbolic and Michelsen respectively. Moreover, considering that the CPU consumes significantly more power, the advantages of using reconfigurable hardware becomes even more apparent. We measured our FPGA architecture to offer an order of magnitude better performance in terms of overall energy efficiency, deploying the optimised accelerator cores, over that of the four-thread CPU. Specifically we measured the energy to solution improvement to reach 13x and 12x for the Hyperbolic and Michelsen algorithms respectively, as shown in Table D.5.

Table D.5: Speedup and efficiency attained using HW accelerator cores

optimized core	**vs. simple**	**vs. 4-thread CPU**	**vs. 1-thread CPU**	**efficiency vs. 4-thread**
Hyperbolic	22.5	1.3	3.8	13
Michelsen	12.5	1.2	3.1	12

Another important outcome, that actually took effect after this first research on the RS algorithms, signified our selection of the Michelsen core as the more preferred one for our succeeding investigation on the UNILOGIC architecture, and although the Hyperbolic seemed slightly better performing up to this point. Both the Michelsen and Hyperbolic kernels have exactly the same structure and only differ in terms of mathematical operations. They actually both solve the exact same problem, as they constitute variations of the N-R (Newton-Raphson) method. They both complete execution when the algorithm converges. However, their main difference is that they converge in a different way. Due to this seemingly slight variation, the optimized Michelsen accelerator implementation proved to require less FPGA resources, and more importantly, to cause less FPGA routing congestion. The latter is highly important when building partial bitstreams, suited for the succeeding UNILOGIC based exploration on dynamic partial reconfiguration. What is more, related to this variation, we later managed to better improve improve the timing for the Michelsen rather than for the Hyperbolic core. This way we could achieve higher clock rates and thus higher execution speeds for

Michelsen. So finally, Michelsen -slightly- outperformed Hyperbolic both in terms of resources, as well as performance.

Appendix E

Projection to Exa-Scale

In order to offer a deeper insight and better understanding on the effectiveness of our approach, we conducted an analysis, targeting the potential to reach the exa-scale performance mark by building on the UNILOGIC architecture. This mainly pertains to an estimation of the hardware resources that would be required if the UNILOGIC system scaled up to offer a performance of 1 ExaFLOPS, and in parallel to an estimation of the related power demands.

Currently, the No.1 HPC machine, i.e. the No.1 supercomputer in the top500 list [112], is the "Summit" by IBM. Interestingly, this same machine scores No.5 in the green500 list [37] with the most energy-efficient supercomputers, boasting a performance that is quite close to that of the No.1 in the same list, the 7nm ARM-based "A64FX prototype". Hence "Summit" supercomputer stands out as the state-of-the-art case, to be used for an aggressive comparison, reaching 148 PFLOPS and consuming no less than 10 MW of power.

TABLE E.1: Projection to Exa-Scale performance, based on accelerated computation on the implemented UNILOGIC prototype technology

Hierarchy	Scale	Performance	Power
MPSoC (Heterogeneous CPU/FPGA Compute Unit)	-	300 GFLOPS	10W
Compute Node	4 × MPSoC	1.2 TFLOPS	50 W
Baseboard	8 × Node	9.6 TFLOPS	400 W
Rack	72 × Baseboard	0.7 PFLOPS	28 KW
HPC System	1400 × Rack	1 ExaFLOPS	39 MW

Regarding our FPGA-based UNILOGIC platform, each FPGA has proved to reach about 280 GFLOPS for the tailored matrix multiplication core. Besides that, the pin-compatible Xilinx ZU15EG device incorporates about 40% more DSP blocks, and thus could end up offering about 400 GFLOPS per device. Nevertheless, we will use a modest estimate, and will move on with our projection considering 300 GFLOPS per FPGA device as the starting mark. Based on this, we can estimate demands to reach exa-scale performance, as reported in table E.1. A compute Node, i.e. a QFDB, would offer 1.2 TFLOPS, and a baseboard, i.e. eight QFDBs, would reach 9.6 TFLOPS. Accordingly, and based on the quite promising scaling characteristics of our architecture, the ExaFLOPS mark would require about 1400 Racks and 39 MW of power.

TABLE E.2: Projection to Exa-Scale performance, based on accelerated computation, as extrapolated to current (not future) FPGA technologies, in a UNILOGIC based prototype

Hierarchy	**Scale**	**Performance**	**Power**
10-12nm Xilinx VU-like FPGA	-	1.5 TFLOPS	20W
Compute Node	4 × FPGA	6 TFLOPS	80 W
Baseboard	8 × Node	48 TFLOPS	640 W
Rack	72 × Baseboard	3.4 PFLOPS	46 KW
HPC System	290 × Rack	1 ExaFLOPS	13 MW

Moving a step further, it is interesting to conduct an extrapolation, using current FPGA technologies. We could even extrapolate more aggressively, referencing advertised forthcoming technologies, however, even a with the modest projection on FPGA technologies already in the market, the respective outcome is quite promising. Initially, looking at the supercomputers on the lists mentioned above, these are based on devices build with fabrication technologies ranging from 12nm down to even TSMC's 7nm process. Our prototype on the other side, hosts FPGAs manufactured with the FinFET+ 16nm process. If we considered an FPGA of similar or even slightly worse technology, i.e. around 10-12nm, which are already in the market as the Intel Agilex 10nm family, this would result to about 60% less power consumption[1]. Furthermore, taking into account currently available

[1]Moving from a 10nm to a 7nm technology leads to 40% less power consumption, while advancing from 7nm to 5nm technology reduces power consumption by 20%. TSMC is already sampling Apple's 5 nm A14 Bionic SoCs for 2020 iPhones

acceleration-targeted devices as the Xilinx Versal family, or even more standardized devices such as the larger Xilinx Virtex Ultrascale+ devices, i.e. the VU13P, we could most probably be able to reach up to about 1.5 TFLOPS per FPGA. Based on this starting point, and as reported through table E.2, we would have to deploy about 290 Racks, while consuming about 13 MW of power. In comparison, a perfect scaling "Summit"-like system, would require about 2,000 Racks and 67 MW of power in order to reach the 1 ExaFLOPS mark.

Summing up our projection to exascale, i.e. to 10^{18} operations per second, we have seen that even considering the currently realized UNILOGIC prototype, we can have a viable scenario to reach 1 ExaFLOPS, while still being more effective compared to state-of-the-art HPC machines. What is more, our prototype is built on MPSoC devices that are already three years old, while a lot of improvement space exists nowadays. Nonetheless, even considering currently available, standard FPGA devices, and extrapolating available performance, we proclaim appealing potential to exascale, outperforming top ranking HPC machines, both it terms of power consumption and size, i.e. required space.

Appendix F

Details on Hypercube Variations

As we have seen in section 4.1.2. the enriched 3D Cube topology, i.e. the 3D Cube with enriched connectivity, corresponds to the interconnection topology of QFDBs on the custom baseboard, and is reproduced in more simple form in Figure F.1 for ease of access. This topology actually corresponds to a Folded HyperCube FHC(n) variation [2, 63, 68] where n is the dimension size equal to 3 for this single cube. In a normal cube, or hypercube in the general case, the diameter has a logarithmic growth.

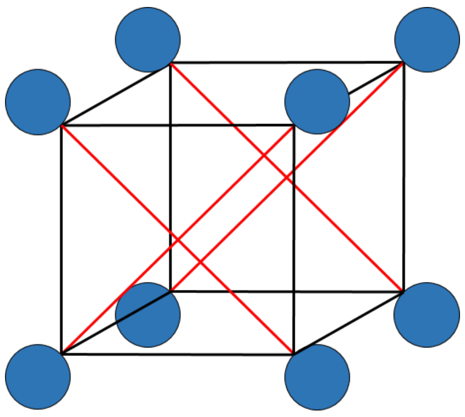


FIGURE F.1: The 3D Cube with enriched connectivity, giving a FHC(3) variation

If we represent the nodes though a binary labeling of the nodes in such a topology, the distance between node A with binary representation b_A and node B with b_B is:

$$Hamming\_weight(b_A \oplus b_B)$$

i.e. the number of different bits between the two numbers. Then, moving on to the mean internode distance, this is defined for a regular network as the ratio of the sum of distances between a node and all other nodes to the total number of nodes:

$$d_a = \sum_{d=1}^{r} \frac{(d.N_d)}{N-1}$$

while in a normal hypercube of dimension n, the distance becomes $n/2$.

To build a FHC(n) as the FHC(3) of Figure F.1, we need to upgrade the node connectivity from n (on a normal cube) to n+1, i.e. add one outgoing link per node. The diameter, i.e. the maximum distance now falls from n to $\lceil n/2 \rceil$, while the average distance falls from $n/2$ to:

$$d_a = \binom{n+1}{\frac{n}{2}+1}$$

which gives 1.25 for n=3 and 1.5 for n=4.

The bisection width is also an important parameter for evaluating the performance of interconnection networks. This is the least number of wires that you should cut in order to divide the network into two equal halves. High bisection is always better. For a normal hypercube HC of dimension n, HC(n), this equals to 2^{n-1}, while for an FHC(n) this equals to 2^n. For 3D cubes this gives an increase from 4 to 8.

List of Publications

F. Chaix, A.D. Ioannou, N. Kossifidis, N. Dimou, G. Ieronymakis, M. Marazakis, V. Papaefstathiou, V. Flouris, M. Ligerakis, G. Ailamakis, T.C. Vavouris, A. Damianakis, M. G.H. Katevenis and I. Mavroidis: "**Implementation and impact of an ultra-compact multi-FPGA board for large system prototyping**", *5th International Workshop on Heterogeneous High-performance Reconfigurable Computing ($H^2RC'19$), held in conjunction with SC'19, 2019* (This paper presents our effort on the QFDB bring-Up and evaluation, as well as many other use cases and achievements based on this board. We also present the core innovation of the UNILOGIC approach, and also many energy efficient measurements stemming purely from the work on this thesis. The stress testing of the QFDB is also reported, as well as a lot of evaluation on FPGA-based acceleration.)

A. Ioannou, P. Malakonakis, K. Georgopoulos, I. Papaefstathiou, A. Dollas and I. Mavroidis: "**Optimized FPGA Implementation of a Compute-Intensive Oil Reservoir Simulation Algorithm**", *Proceedings of the 19th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), July 7-11, 2019* (This paper presents a basic portion of this thesis' research on the UNILOGIC architecture, and focusing on co-design and optimization for efficient acceleration, when deploying single or multiple parallel accelerators, on FPGAs of the QFDB prototype.)

K. Georgopoulos, K. Bakanov, I. Mavroidis, I. Papaefstathiou, A. Ioannou, P. Malakonakis, K. Pham, D. Koch, L. Lavagno: "**A Novel Framework for Utilising Multi-FPGAs in HPC Systems**", *Chapter in Heterogeneity Alliance Book "Heterogeneous Computing Architectures", 2019* (This chapter presents the technology developed in the context of European project ECOSCALE, with the embodied architecture developed and implemented as part of my thesis. It aims at contributing to the effort for a state-of-the-art platform that can satisfactorily serve applications in the context of High Performance Computing (HPC). The platform is presented from both software framework aspect, that allows users to introduce their applications described in OpenCL, as well from the architectural aspect, that utilises reconfigurable hardware in order to execute at high speeds computationally-intensive tasks. The Worker within the UNILOGIC is explained, featuring all elements required for a single computational component, i.e. a processing system, memory and reconfigurable logic. The latter hosts the computationally-intensive tasks, referred to as accelerator modules. Furthermore, the complete prototype is presented, comprised of 64 FPGAs (Workers), which thanks to UNILOGIC technology, will be available for exploitation by any given application in a seamless and transparent manner. The added benefit from using FPGA technology to potentially reach a low-power HPC system is also explained.)

G. Pitsis, G. Tsagkatakis, C. Kozanitis, I. Kalomoiris, A. Ioannou, A. Dollas, M. Katevenis, and P. Tsakalides: **"Efficient Convolutional Neural Network Weight Compression for Space Data Classification on Multi-FPGA Platforms"**, *in Proc. 44th International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2019), Brighton, UK, May 12-17, 2019* (The power efficient FPGA alternative for on-satellite deep learning is presented. It constitutes a good use case for (part of) my thesis-related work. It also validates the hardware prototypes based on the QFDB board, which involves my contribution. I was also involved on the implementation of the hardware architecture, which furthermore includes ideas stemming from the core of my thesis-related work on UNILOGIC.)

I. Kalomoiris, G. Pitsis, G. Tsagkatakis, A. Ioannou, C. Kozanitis, A. Dollas, P. Tsakalides, and M. Katevenis: **"An Experimental Analysis of the Opportunities to Use Field Programmable Gate Array Multiprocessors for On-board Satellite Deep Learning Classification of Spectroscopic Observations from Future ESA Space Missions"**, *in Proc. 2019 European Workshop on On-Board Data Processing (OBDP 2019), European Space Research and Technology Centre (ESTEC), The Netherlands, February 25-27, 2019* (An FPGA based architecture for on-satellite deep learning is presented. It constitutes a good use case for basic aspects of my thesis-related work. The hardware platform that verifies the specified technology proposed is run on the Quad-FPGA boards, on which I contributed. I was also involved on the software application running in order to have single and then multiple FPGAs operating on data, and also on various hardware design aspects and architectural configurations, with these solutions derived from my thesis work.)

Aggelos Ioannou, Konstantinos Georgopoulos, Pavlos Malakonakis, Iakovos Mavroidis, Ioannis Papaefstathiou: **"ECOSCALE – Towards HPC based on the UNILOGIC architecture and a novel interconnect/addressing scheme for global resource sharing"**, *Towards European Exascale HPC Workshop at the HiPEAC'19 Conference, Valencia, Jan. 2019* (This publication presents the efforts entailed in this thesis for effectively realizing the UNILOGIC unification infrastructure, by deploying the PGAS model through properly encompassing AXI interconnects)

F. Chaix, A. Ioannou, N. Kossifidis, M. Ligerakis, I, Mavroidis, M. Katevenis: **"Experience on the ongoing Bring-up of ExaNeSt's ZYNQ Ultrascale+ -based high density daughter board"**, *Towards European Exascale HPC Workshop at the HiPEAC'19 Conference, Valencia, Jan. 2019* (Presented by the author of this thesis, and referenced by "The Next Platform" article "HiPEAC: Shifting Focal Points in European HPC Research", https://www.nextplatform.com/2019/01/22/hipeac-shifting-focal-points-in-hpc-research/ [39]. It focuses on the efforts to expedite the QFDB (the Quad-FPGA daughter board used by ECOSCALE and collaborating projects) bring-Up and validation. The QFDB is used as a basic building block of the prototype used in this thesis. Includes a lot of testing and also specially designed hardware that uses techniques derived during this thesis, in order to allow a RAM-less Linux boot, i.e. booting Linux in the absence of local memory, and borrowing all the DDR memory from a remote donor node. All nodes incorporate hardware specially designed by the author of this thesis. Also, it contributes an IP for configurable FPGA stress testing, on which I was involved)

A. Ioannou, P. Malakonakis, K. Georgopoulos, I. Papaefstathiou, I. Mavroidis, A. Dollas: **"FPGA accelerator optimizations for Diversified Oil Reservoir Simulation Algorithms"**, *Contest winner in: International contest on Accelerated Heterogeneous Cloud Computing, organized within AccelCloud: Workshop and Contest on Accelerating Big Data in Heterogeneous Cloud computing, HiPEAC'19 Conference, Valencia, Jan. 2019* (This work presents the research on co-design and optimization for efficient acceleration, when deploying the UNILOGIC architecture, with single or multiple parallel accelerators, on the QFDB platform's FPGAs.)

Yann Beilliard, Maxime Godard, Aggelos Ioannou, Astrinos Damianakis, Michael Ligerakis, Iakovos Mavroidis, Pierre-Yves Martinez, David Danovitch, Julien Sylvestre, Dominique Drouin: "**FPGA-based Multi-Chip Module for High-Performance Computing - ExaNoDe**", *Towards European Exascale HPC Workshop at the HiPEAC'19 Conference, Valencia, Jan. 2019* (The efforts on the ExaNoDe building block, which is analogous to the QFDB, is presented. A lot of my efforts are entailed on this work, on delivering a state-of-the-art multi-chip module, for High-Performance Computing based on reconfigurable resources.)

Pavlos Malakonakis, Konstantinos Georgopoulos, Aggelos Ioannou, Luciano Lavagno, Ioannis Papaefstathiou, Iakovos Mavroidis: "**HLS Algorithmic Explorations for HPC Execution on Reconfigurable Hardware – ECOSCALE**", *Applied Reconfigurable Computing. Architectures, Tools, and Applications - 14th International Symposium, ARC 2018* (This paper documents on the exploration for two algorithms in the field of oil Reservoir Simulation (RS). These are encompassed on the UNILOGIC architecture of this thesis. This h/w architecture realization on the FPGAs, is the core of my work, offering the uniform global addressing, scheduling and parallelism. Includes also efforts on comprising the s/w bare-metal testing and executing them on the MPSoC, gathering results and respinning with cycles of design optimizations targeting on the weaknesses spotted, until we reached the favorable results. Also gathered basic energy efficiency results.)

Pavlos Malakonakis, Konstantinos Georgopoulos, Aggelos Ioannou, Luciano Lavagno, Ioannis Papaefstathiou and Iakovos Mavroidis: "**HLS Algorithmic Explorations for HPC Execution on Reconfigurable Hardware**", *PRACEdays18, European HPC Summit Week, May 2018* (This reports on the co-design process of optimizing the UNILOGIC architecture along with the HLS process on building efficient accelerators.)

Konstantinos Georgopoulos, Angelos Ioannou, Pavlos Malakonakis, Iakovos Mavroidis, Vassilis Papaefstathiou, Ioannis Sourdis: "**UNIMEM and UNILOGIC architectures for local/remote sharing of resources**", *ExascaleHPC Workshop, at the HiPEAC'18 Conference, Manchester, Jan. 2018* (This publication focuses on the merging of the known UNIMEM architecture, along with the innovative UNILOGIC architecture, of which the designing, combining and implementation is the core of my thesis.)

Iakovos Mavroidis, Aggelos Ioannou and Konstantinos Georgopoulos: "**Injecting and managing accelerators: ECOSCALE (Energy-Efficient Heterogeneous Computing at Exascale)**", *HiPEACinfo magazine, October 2017* (This article describes the targets of the UNILOGIC approach within the UNILOGIC project, with the core of this architecture pertaining to this thesis realization of the core architecture in hardware.)

Accepted[1]:

Aggelos D. Ioannou, Konstantinos Georgopoulos, Pavlos Malakonakis, Ioannis Papaefstathiou, Vassilis Papaefstathiou, Dionisios N. Pnevmatikatos and Iakovos Mavroidis: "**UNILOGIC: A novel architecture for Highly Parallel Reconfigurable Systems**", *ACM Transactions on Reconfigurable Technology and Systems (TRETS) journal* (This paper presents the core of my thesis work, elaborating on both the novel UNILOGIC approach, as well as on the implementation and evaluation on a real life hardware platform, incorporating 64 FPGAs, with all of them jointly orchestrated and operated in parallel.)

In preparation for Submission:

George Pitsis, Christos Kozanitis, Aggelos Ioannou, Grigorios Tsagkatakis, Charisios Loukas, Apostolos Dollas, Manolis G.H. Katevenis, Panagiotis Tsakalides: "**Efficient Convolutional Neural Network Weight Compression with Inter-Stage Pipelining on Multi-FPGA Platforms**", *ACM Transactions on Architecture and Code Optimization (TACO) journal* (This paper thoroughly presents the work for on-satellite deep learning based on the power efficient FPGA architecture. It constitutes a proof of concept use case for (part of) my thesis-related work. It also validates the hardware prototypes based on the QFDB board. Ideas stemming from the core of my thesis-related work is entailed, on which I was involved on both the the hardware architecture, as well as the implementation and verification.)

[1]recommended for publication with minor revisions

References

[1] Milton Abramowitz. *Handbook of Mathematical Functions, With Formulas, Graphs, and Mathematical Tables,* New York, NY, USA: Dover Publications, Inc., 1974. ISBN: 0486612724.

[2] Nibedita Adhikari and Dr Tripathy. "The Folded Crossed Cube : A New Interconnection Network For Parallel Systems". In: *International Journal of Computer Applications* 4 (July 2010). DOI: 10.5120/807-1147.

[3] Amran Al-Aghbari and Muhammad Elrabaa. "A Platform for FPGA Virtualization in Clouds and Data Centers". In: *Microprocessors and Microsystems* 62 (July 2018). DOI: 10.1016/j.micpro.2018.07.010.

[4] Roberto Ammendola, Andrea Biagioni, Paolo Cretaro, et al. "The Next Generation of Exascale-Class Systems: The ExaNeSt Project". In: *Euromicro Conference on Digital System Design, DSD 2017, Vienna, Austria, August 30 - Sept. 1, 2017*. 2017, pp. 510–515. DOI: 10.1109/DSD.2017.20. URL: https://doi.org/10.1109/DSD.2017.20.

[5] Arslan Arif, Felipe A. Barrigon, Francesco Gregoretti, et al. "Performance and energy-efficient implementation of a smart city application on FPGAs". In: *Journal of Real-Time Image Processing* (2018), pp. 1–15.

[6] Xilinx Avnet. *MicroZed / Zedboard*. 2019. URL: http://zedboard.org/product/microzed.

[7] *AXI Reference Guide*. 2017. URL: www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.

[8] Ayoosh Bansal, Rohan Tabish, Giovani Gracioli, et al. "Evaluating the Memory Subsystem of a Configurable Heterogeneous MPSoC". In: July 2018.

[9] *BEEcube FPGA Based Rapid Prototyping Platforms*. 2016. URL: http://docplayer.net/33010877-Beecube-fpga-based-rapid-prototyping-platforms-for-military-communications.html.

[10] Yann Beilliard, Maxime Godard, Aggelos Ioannou, et al. “FPGA-based Multi-Chip Module for High-Performance Computing”. In: *CoRR* abs/1906.11175 (2019). arXiv: 1906.11175. URL: http://arxiv.org/abs/1906.11175.

[11] BittWare. *BittWare FPGA Acceleration*. 2019. URL: https://www.bittware.com/.

[12] Michaela Blott. “Reconfigurable future for HPC”. In: *2016 International Conference on High Performance Computing Simulation (HPCS)*. 2016, pp. 130–131. DOI: 10.1109/HPCSim.2016.7568326.

[13] Brad Brech, Juan Rubio, and Michael Hollinger. *Data Engine for NoSQL - IBM Power Systems Edition*. White Paper. 2015.

[14] F. Chaix, A.D. Ioannou, N. Kossifidis, et al. “Implementation and impact of an ultra-compact multi-FPGA board for large system prototyping”. In: *5th International Workshop on Heterogeneous High-performance Reconfigurable Computing (H^2RC'19), held in conjunction with SC'19*. 2019.

[15] Alessandro Cilardo. “HtComp: bringing reconfigurable hardware to future high-performance applications”. In: *IJHPCN* 12.1 (2018), pp. 74–83. DOI: 10.1504/IJHPCN.2018.10015028. URL: https://doi.org/10.1504/IJHPCN.2018.10015028.

[16] ASC Community. “Network Communication in a Supercomputing System”. In: *The Student Supercomputer Challenge Guide: From Supercomputing Competition to the Next HPC Generation*. Singapore: Springer Singapore, 2018, pp. 41–60. ISBN: 978-981-10-3731-3. DOI: 10.1007/978-981-10-3731-3_3. URL: https://doi.org/10.1007/978-981-10-3731-3_3.

[17] Jason Cong, Zhenman Fang, Michael Lo, et al. “Understanding Performance Differences of FPGAs and GPUs”. In: Apr. 2018, pp. 93–96. DOI: 10.1109/FCCM.2018.00023.

[18] Convey Computer Corp. *The Convey HC-2 Computer Architectural Overview (White Paper)*. https://www.micron.com/-/media/documents/products/white-paper/wp_convey_hc2_architectual_overview.pdf. 2012.

[19] Intel Corp. *Intel Agilex FPGA Advanced Information Brief*. 2019. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/agilex/ag-overview.pdf.

[20] Roberto Sanchez Correa and Jean-Pierre David. “Ultra-low latency communication channels for FPGA-based HPC cluster”. In: *Integration* 63 (2018), pp. 41–55. DOI: 10.1016/j.vlsi.2018.05.005. URL: https://doi.org/10.1016/j.vlsi.2018.05.005.

[21] Philippe Coussy and Adam Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuits*. June 2008.

[22] Yves Durand, Paul M. Carpenter, Stefano Adami, et al. "EUROSERVER: Energy Efficient Node for European Micro-Servers". In: *17th Euromicro Conference on Digital System Design, DSD 2014, Verona, Italy, August 27-29, 2014*. 2014, pp. 206–213. DOI: 10.1109/DSD.2014.15. URL: https://doi.org/10.1109/DSD.2014.15.

[23] F. A. Escobar, X. Chang, and C. Valderrama. "Suitability Analysis of FPGAs for Heterogeneous Platforms in HPC". In: *IEEE Transactions on Parallel and Distributed Systems* 27.2 (2016), pp. 600–612. ISSN: 1045-9219. DOI: 10.1109/TPDS.2015.2407896.

[24] EU. *The EuroEXA Project*. https://euroexa.eu/. 2017-2020.

[25] EU. *The Euroserver Project*. http://www.euroserver-project.eu. 2013-2017.

[26] Umer Farooq, Roselyne Chotin-Avot, Moazam Azeem, et al. "Inter-FPGA Routing Environment for Performance Exploration of multi-FPGA Systems". In: *Proceedings of the 27th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype*. RSP '16. Pittsburgh, Pennsylvania: ACM, 2016, pp. 107–113. ISBN: 978-1-4503-4535-4. DOI: 10.1145/2990299.2990317. URL: http://doi.acm.org/10.1145/2990299.2990317.

[27] Michael Feldman. "Exascale Is Not Your Grandfather's HPC". In: *The Next Platform, Stackhouse Publishing Inc* (2019). URL: https://www.nextplatform.com/2019/10/22/exascale-is-not-your-grandfathers-hpc/.

[28] Johannes de Fine Licht, Simon Meierhans, and Torsten Hoefler. "Transformations of High-Level Synthesis Codes for High-Performance Computing". In: *ArXiv* abs/1805.08288 (2018).

[29] Sandro Fiore, Mohamed Bakhouya, and Waleed Smari. "On the road to exascale: Advances in High Performance Computing and Simulations - An overview and editorial". In: *Future Generation Computer Systems* 82 (May 2018), pp. 450–458. DOI: 10.1016/j.future.2018.01.034.

[30] Kermin Fleming and Michael Adler. "The LEAP FPGA Operating System". In: *FPGAs for Software Programmers*. 2016, pp. 245–258. DOI: 10.1007/978-3-319-26408-0_14. URL: https://doi.org/10.1007/978-3-319-26408-0_14.

[31] Raspberry Pi Foundation. *Teach, Learn, and Make with Raspberry Pi.* 2019. URL: https://www.raspberrypi.org/.

[32] Jeremy Fowers, Greg Brown, Patrick Cooke, et al. "A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-Window Applications". In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays.* FPGA '12. Monterey, California, USA: Association for Computing Machinery, 2012, 47–56. ISBN: 9781450311557. DOI: 10.1145/2145694.2145704. URL: https://doi.org/10.1145/2145694.2145704.

[33] A. D. George, M. C. Herbordt, H. Lam, et al. "Novo-G#: Large-scale reconfigurable computing with direct and programmable interconnects". In: *2016 IEEE High Performance Extreme Computing Conference (HPEC).* 2016, pp. 1–7. DOI: 10.1109/HPEC.2016.7761639.

[34] PRO DESIGN Electronic GmbH. *profpga: FPGA Prototyping.* 2019. URL: https://www.profpga.com.

[35] SciEngines GmbH. *SciEngines Hardware, High Performance Reconfigurable Computing.* 2019. URL: https://www.sciengines.com/technology-platform/sciengines-hardware/.

[36] Trenz Electronic GmbH. *Technical Reference Manual of Trenz TEBF0808.* 2019. URL: https://wiki.trenz-electronic.de/display/PD/TEBF0808+TRM.

[37] *Green500 List - November 2019.* 2019. URL: www.top500.org/green500/list/2019/11/.

[38] Nicolae Bogdan Grigore, Charalampos Kritikakis, and Dirk Koch. "HLS Enabled Partially Reconfigurable Module Implementation". In: *Architecture of Computing Systems - ARCS 2018 - 31st International Conference, Braunschweig, Germany, April 9-12, 2018, Proceedings.* 2018, pp. 269–282. DOI: 10.1007/978-3-319-77610-1_20. URL: https://doi.org/10.1007/978-3-319-77610-1_20.

[39] Nicole Hemsoth. *HiPEAC: Shifting Focal Points in European HPC Research.* 2019. URL: www.nextplatform.com/2019/01/22/hipeac-shifting-focal-points-in-hpc-research/.

[40] Edson Horta, Xinzi Shen, Khoa Pham, et al. "Accelerating Linux Bash Commands on FPGAs Using Partial Reconfiguration". In: *Proceedings of FPGAs for Software Programmers (FSP 2017) conference.* Oct. 2017.

[41] Network of Expertise HPC Advisory Council. *Interconnect Analysis: 10GigE and InfiniBand in High Performance Computing*. White Paper. 2009. URL: https://www.hpcadvisorycouncil.com/pdf/IB_and_10GigE_in_HPC.pdf.

[42] Muhuan Huang, Di Wu, Cody Hao Yu, et al. "Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*. 2016, pp. 456–469. DOI: 10.1145/2987550.2987569. URL: https://doi.org/10.1145/2987550.2987569.

[43] Ron Huizen. *FPGAs in the Cloud: Should you Rent or Buy FPGAs for Development and Deployment?* White Paper. 2018. URL: www.bittware.com/resources/fpgas-in-the-cloud.

[44] Amazon.com Inc. *Amazon EC2 F1 Instances*. 2019. URL: https://aws.amazon.com/ec2/instance-types/f1/.

[45] Digilent Inc. *FPGA, Microcontrollers and Instrumentation*. 2019. URL: http://www.digilent.com.

[46] Maxeler Technologies Inc. *Dataflow Computing*. 2019. URL: https://www.maxeler.com/technology/dataflow-computing/.

[47] Maxeler Technologies Inc. *Maxeler Products*. 2019. URL: https://www.maxeler.com/products/.

[48] Xilinx Inc. *7 Series FPGAs Configuration*. 2018. URL: www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf.

[49] Xilinx Inc. *Versal Architecture and Product Data Sheet: Overview*. 2019. URL: https://www.xilinx.com/support/documentation/data_sheets/ds950-versal-overview.pdf.

[50] National Instruments. *Automated Test and Automated Measurement Systems*. 2019. URL: http://www.ni.com/en-us/innovations/wireless/software-defined-radio.html.

[51] Aggelos D. Ioannou, Pavlos Malakonakis, Konstantinos Georgopoulos, et al. "Optimized FPGA Implementation of a Compute-Intensive Oil Reservoir Simulation Algorithm". In: *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Springer International Publishing, Aug. 2019, pp. 442–454. ISBN: 978-3-030-27561-7. DOI: 10.1007/978-3-030-27562-4_32.

[52] A. Iordache, G. Pierre, P. Sanders, et al. “High performance in the cloud with FPGA groups”. In: *Proceedings of the 9th International Conference on Utility and Cloud Computing, UCC 2016, Shanghai, China, December 6-9, 2016*. 2016, pp. 1–10. DOI: 10.1145/2996890.2996895. URL: https://doi.org/10.1145/2996890.2996895.

[53] Chris Kachris. *Scalable FPGA-based Accelerators on the Cloud*. https://web.fe.up.pt/~specs/events/wrc2019/presentations/Chris_Kachris_ Inaccel.pdf. 2019.

[54] Christoforos Kachris, Babak Falsafi, and Dimitrios Soudris. *Hardware Accelerators in Data Centers*. Jan. 2019. ISBN: 978-3-319-92791-6. DOI: 10.1007/978-3-319-92792-3.

[55] Christoforos Kachris, Dimitrios Soudris, Georgi Gaydadjiev, et al. “The VINEYARD Approach: Versatile, Integrated, Accelerator-Based, Heterogeneous Data Centres”. In: *Proceedings of the 12th International Symposium on Applied Reconfigurable Computing - Volume 9625*. Berlin, Heidelberg: Springer-Verlag, 2016, 3–13. ISBN: 9783319304809. DOI: 10.1007/978-3-319-30481-6_1. URL: https://doi.org/10.1007/978-3-319-30481-6_1.

[56] Christoforos Kachris, Hans Vandierendonck, Dimitrios Nikolopoulos, et al. “The VINEYARD integrated framework for hardware accelerators in the cloud”. In: July 2018, pp. 236–243. DOI: 10.1145/3229631.3236093.

[57] Ioannis Kalomoiris, George Pitsis, Grigorios Tsagkatakis, et al. “An Experimental Analysis of the Opportunities to Use Field Programmable Gate Array Multiprocessors for On-board Satellite Deep Learning Classification of Spectroscopic Observations from Future ESA Space Missions”. In: *On-Board Data Processing (OBDP)*. Feb. 2019.

[58] Eunsuk Kang, Ethan Jackson, and Wolfram Schulte. “An Approach for Effective Design Space Exploration”. In: vol. 6662. Mar. 2010, pp. 33–54. DOI: 10.1007/978-3-642-21292-5_3.

[59] N. Kapre and J. Gray. “Hoplite: A Deflection-Routed Directional Torus NoC for FPGAs”. In: *TRETS* 10.2 (2017), 14:1–14:24. DOI: 10.1145/3027486. URL: https://doi.org/10.1145/3027486.

[60] A. Kashif and M. A. S. Khalid. “Experimental evaluation and comparison of time-multiplexed multi-FPGA routing architectures”. In: *2016 IEEE 59th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2016, pp. 1–4.

[61] M. Katevenis. “Interprocessor Communication seen as Load-Store Instruction Generalization”. In: *The Future of Computing, essays in memory of Stamatis Vassiliadis*. K. Bertels e.a. Editors, Delft, The Netherlands, 28 Sep. 2007, 2007, pp. 55–68.

[62] Manolis Katevenis, Nikolaos Chrysos, Manolis Marazakis, et al. “The ExaNeSt Project: Interconnects, Storage, and Packaging for Exascale Systems”. In: *2016 Euromicro Conference on Digital System Design, DSD 2016, Limassol, Cyprus, August 31 - September 2, 2016*. 2016, pp. 60–67. DOI: 10.1109/DSD.2016.106. URL: https://doi.org/10.1109/DSD.2016.106.

[63] Zaki Khan. “Topological Evaluation of Variants Hypercube Network”. In: *Asian journal of computer science and information technology* 3 (Sept. 2013), p. 9.

[64] Ryohei Kobayashi, Yuma Oobata, Norihisa Fujita, et al. “OpenCL-ready High Speed FPGA Network for Reconfigurable High Performance Computing”. In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2018, Chiyoda, Tokyo, Japan, January 28-31, 2018*. 2018, pp. 192–201. DOI: 10.1145/3149457.3149479. URL: https://doi.org/10.1145/3149457.3149479.

[65] Dirk Koch. *Partial Reconfiguration on FPGAs – Architectures, Tools and Applications*. New York, NY, USA: Springer-Verlag New York, 2012. ISBN: 0486612724.

[66] Jens Korinth, Jaco Hofmann, Carsten Heinz, et al. “The TaPaSCo Open-Source Toolflow for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems”. In: Jan. 2019, pp. 214–229. ISBN: 978-3-030-14801-0. DOI: 10.1007/978-3-030-17227-5_16.

[67] Douglas Kothe, Stephen Lee, and Irene Qualters. “Exascale Computing in the United States”. In: *Computing in Science and Engineering* 21 (Jan. 2019), pp. 17–29. DOI: 10.1109/MCSE.2018.2875366.

[68] S. Latifi. “Hypercube-based topologies with incremental link redundancy”. In: (Jan. 1989).

[69] James Laudon and Daniel Lenoski. “The SGI Origin: A ccNUMA Highly Scalable Server”. In: *Proceedings of the 24th International Symposium on Computer Architecture, Denver, Colorado, USA, June 2-4, 1997*. 1997, pp. 241–251. DOI: 10.1145/264107.264206. URL: http://doi.acm.org/10.1145/264107.264206.

[70] Yangfan Liu, Peng Liu, Yingtao Jiang, et al. “Building a multi-FPGA-based emulation framework to support networks-on-chip design and verification”. In: *International Journal of Electronics - INT J ELECTRON* 97 (Oct. 2010), pp. 1241–1262. DOI: 10.1080/00207217.2010.512017.

[71] HiTech Global LLC. *Xilinx / Altera FPGA boards, design services, & IP Cores.* 2019. URL: http://www.hitechglobal.com/.

[72] Spyros Lyberis, George Kalokerinos, Michalis Lygerakis, et al. “FPGA Prototyping of Emerging Manycore Architectures for Parallel Programming Research using Formic Boards”. In: *Journal of Systems Architecture* 60 (June 2014). DOI: 10.1016/j.sysarc.2014.03.002.

[73] G. Mahesh and Sakthivel SM. “Verification of memory transactions in AXI protocol using system verilog approach”. In: *2015 International Conference on Communications and Signal Processing (ICCSP).* 2015, pp. 0860–0864.

[74] Mariem Makni, Mouna Baklouti, Smail Niar, et al. “Performance Exploration of AMBA AXI4 Bus Protocols for Wireless Sensor Networks”. In: Oct. 2017. DOI: 10.1109/AICCSA.2017.26.

[75] Pavlos Malakonakis, Konstantinos Georgopoulos, Aggelos Ioannou, et al. “HLS Algorithmic Explorations for HPC Execution on Reconfigurable Hardware - ECOSCALE”. In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications - 14th International Symposium, ARC 2018, Santorini, Greece, May 2-4, 2018, Proceedings.* 2018, pp. 724–736. DOI: 10.1007/978-3-319-78890-6_58. URL: https://doi.org/10.1007/978-3-319-78890-6_58.

[76] Kristiyan Manev, Anuj Vaishnav, and Dirk Koch. *Unexpected Diversity: Quantitative Memory Analysis for Zynq UltraScale+ Systems.* Oct. 2019. DOI: 10.13140/RG.2.2.12786.66244.

[77] Manolis Marazakis, John Goodacre, Didier Fuin, et al. “EUROSERVER: Share-anything Scale-out Micro-server Design”. In: *Proceedings of the 2016 Conference on Design, Automation & Test in Europe.* DATE ’16. Dresden, Germany: EDA Consortium, 2016, pp. 678–683. ISBN: 978-3-9815370-6-2. URL: http://dl.acm.org/citation.cfm?id=2971808.2971966.

[78] Tomislav Matic, Ivan Aleksi, and Zeljko Hocenski. “CPU, GPU and FPGA implementations of MALD: Ceramic tile surface defects detection algorithm”. In: *Automatika* 55 (Feb. 2014). DOI: 10.7305/automatika.2014.01.317.

[79] Iakovos Mavroidis, Ioannis Papaefstathiou, Luciano Lavagno, et al. "ECOSCALE: Reconfigurable computing and runtime system for future exascale systems". In: *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*. 2016, pp. 696–701. URL: http://ieeexplore.ieee.org/document/7459398/.

[80] Pingfan Meng, Matt Jacobsen, Motoki Kimura, et al. "Hardware accelerated novel optical de novo assembly for large-scale genomes". In: Sept. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927499.

[81] Antoniette Mondigo, Tomohiro Ueno, Daichi Tanaka, et al. "Design and scalability analysis of bandwidth-compressed stream computing with multiple FPGAs". In: *12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip, ReCoSoC 2017, Madrid, Spain, July 12-14, 2017*. 2017, pp. 1–8. DOI: 10.1109/ReCoSoC.2017.8016148. URL: https://doi.org/10.1109/ReCoSoC.2017.8016148.

[82] Syed Waqar Nabi and Wim Vanderbauwhede. "FPGA design space exploration for scientific HPC applications using a fast and accurate cost model based on roofline analysis". In: *Journal of Parallel and Distributed Computing* 133 (2019), pp. 407 –419. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2017.05.014. URL: http://www.sciencedirect.com/science/article/pii/S074373151730165X.

[83] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, et al. "GraphGen: An FPGA Framework for Vertex-Centric Graph Computation". In: May 2014, pp. 25–28. ISBN: 978-1-4799-5111-6. DOI: 10.1109/FCCM.2014.15.

[84] *OpenCL overview*. 2019. URL: www.khronos.org/opencl/.

[85] Jian Ouyang, Shiding Lin, Wei Qi, et al. "SDA: Software-defined accelerator for large-scale DNN systems". In: *2014 IEEE Hot Chips 26 Symposium (HCS), Cupertino, CA, USA, August 10-12, 2014*. 2014, pp. 1–23. DOI: 10.1109/HOTCHIPS.2014.7478821. URL: https://doi.org/10.1109/HOTCHIPS.2014.7478821.

[86] Kyprianos Papadimitriou, Apostolos Dollas, and Scott Hauck. "Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model". In: *TRETS* 4 (Dec. 2011), p. 36. DOI: 10.1145/2068716.2068722.

[87] Oliver Pell and Vitali Averbukh. "Maximum Performance Computing with Dataflow Engines". In: *Computing in Science and Engineering* 14.4 (2012), pp. 98–103. DOI: 10.1109/MCSE.2012.78. URL: https://doi.org/10.1109/MCSE.2012.78.

[88] Oliver Pell and Oskar Mencer. “Surviving the End of Frequency Scaling with Reconfigurable Dataflow Computing”. In: *SIGARCH Comput. Archit. News* 39.4 (Dec. 2011), pp. 60–65. ISSN: 0163-5964. DOI: 10.1145/2082156.2082172. URL: http://doi.acm.org/10.1145/2082156.2082172.

[89] Khoa Pham, Edson Horta, and Dirk Koch. “BITMAN: A tool and API for FPGA bitstream manipulations”. In: Design, Automation and Test in Europe (DATE) ; Conference date: 27-03-2017 Through 31-03-2017. 2017. DOI: 10.23919/DATE.2017.7927114. URL: https://www.date-conference.com/conference/event-overview.

[90] Khoa Pham, Anuj Vaishnav, Malte Vesper, et al. “ZUCL: A ZYNQ UltraScale+ Framework for OpenCL HLS Applications”. In: Sept. 2018.

[91] Khoa Dang Pham, Edson L. Horta, Dirk Koch, et al. “IPRDF: An Isolated Partial Reconfiguration Design Flow for Xilinx FPGAs”. In: *12th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip, MCSoC 2018, Hanoi, Vietnam, September 12-14, 2018*. 2018, pp. 36–43. DOI: 10.1109/MCSoC2018.2018.00018. URL: https://doi.org/10.1109/MCSoC2018.2018.00018.

[92] George Pitsis, Grigorios Tsagkatakis, Christos Kozanitis, et al. “Efficient Convolutional Neural Network Weight Compression for Space Data Classification on Multi-fpga Platforms”. In: *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2019, pp. 3917–3921. DOI: 10.1109/ICASSP.2019.8682732.

[93] Christian Plessl. “Bringing FPGAs to HPC Production Systems and Codes”. In: *Fourth International Workshop on Heterogeneous High-performance Reconfigurable Computing*. workshop at Supercomputing. 2018.

[94] Manolis Ploumidis, Nikolaos Kallimanis, Marios Asiminakis, et al. “Software and Hardware co-design for low-power HPC platforms”. In: *5th International Workshop on Communication Architectures for HPC, Big Data, Deep Learning and Clouds at Extreme Scale (ExaComm’19), Frankfurt, Germany, June 20, 2019*. 2019. DOI: 10.1109/DSD.2016.106. URL: https://doi.org/10.1109/DSD.2016.106.

[95] Danny Price, M. Clark, B. Barsdell, et al. “Optimizing performance per watt on GPUs in High Performance Computing: temperature, frequency and voltage effects”. In: *Computer Science - Research and Development* (July 2014). DOI: 10.1007/s00450-015-0300-5.

[96] BERTEN Digital Signal Processing. *GPU vs FPGA Performance Comparison.* http://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf. 2016.

[97] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, et al. "A reconfigurable fabric for accelerating large-scale datacenter services". In: *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, USA, June 14-18.* 2014, pp. 13–24. DOI: 10.1109/ISCA.2014.6853195. URL: https://doi.org/10.1109/ISCA.2014.6853195.

[98] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, et al. "A reconfigurable fabric for accelerating large-scale datacenter services". In: *Commun. ACM* 59.11 (2016), pp. 114–122. DOI: 10.1145/2996868. URL: https://doi.org/10.1145/2996868.

[99] *quad SG-280 Prototyping System.* 2017. URL: www.profpga.com/files/profpga_quadsg280_product_brief_3.pdf.

[100] Alvise Rigo, Christian Pinto, Kevin Pouget, et al. "Paving the Way Towards a Highly Energy-Efficient and Highly Integrated Compute Node for the Exascale Revolution: The ExaNoDe Approach". In: *Euromicro Conference on Digital System Design, DSD 2017, Vienna, Austria, August 30 - Sept. 1, 2017.* 2017, pp. 486–493. DOI: 10.1109/DSD.2017.37. URL: https://doi.org/10.1109/DSD.2017.37.

[101] John Russell. "Exascale Watch: El Capitan Will Use AMD CPUs GPUs to Reach 2 Exaflops". In: *HPCwire, Tabor Communications* (2020). URL: https://www.hpcwire.com/2020/03/04/exascale-watch-el-capitan-will-use-amd-cpus-gpus-to-reach-2-exaflops/.

[102] Ravi S, Ezra K, and H Kittur. "Design of a Bus Monitor for Performance Analysis of AXI Protocol based SoC Systems". In: *International Journal of Applied Engineering Research* 9 (Nov. 2014), pp. 6313–6324.

[103] Pradeep S R. "Design and Verification Environment for AMBA AXI Protocol for SoC Integration". In: *International Journal of Research in Engineering and Technology* 03 (May 2014), pp. 338–343. DOI: 10.15623/ijret.2014.0315066.

[104] O. Sander, S. Baehr, E. Luebbers, et al. "A flexible interface architecture for reconfigurable coprocessors in embedded multicore systems using PCIe Single-root I/O virtualization". In: *2014 International Conference on Field-Programmable Technology (FPT).* 2014, pp. 223–226. DOI: 10.1109/FPT.2014.7082780.

[105] Thomas Skordas. "Toward a European Exascale Ecosystem: The EuroHPC Joint Undertaking". In: *Commun. ACM* 62.4 (Mar. 2019), p. 70. ISSN: 0001-0782. DOI: 10.1145/3312567. URL: https://doi.org/10.1145/3312567.

[106] Balaji Subramaniam, Winston Saunders, Tom Scogland, et al. "Trends in energy-efficient computing: A perspective from the Green500". In: *International Green Computing Conference, IGCC 2013, Arlington, VA, USA, June 27-29, 2013, Proceedings*. 2013, pp. 1–8. DOI: 10.1109/IGCC.2013.6604520. URL: https://doi.org/10.1109/IGCC.2013.6604520.

[107] Qingshan Tang. "Methodology of Multi-FPGA Prototyping Platform Generation". NNT: 2015PA066016. PhD thesis. Université Pierre et Marie Curie - Paris, Jan. 2015. URL: https://tel.archives-ouvertes.fr/tel-01256510/document.

[108] Qingshan Tang and Matthieu Tuna. "Performance Comparison between Multi-FPGA Prototyping Platforms: Hardwired Off-the-Shelf, Cabling, and Custom". In: May 2014, pp. 125–132. ISBN: 978-1-4799-5111-6. DOI: 10.1109/FCCM.2014.44.

[109] Naif Tarafdar, Thomas Lin, Eric Fukuda, et al. "Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA: ACM, 2017, pp. 237–246. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021742. URL: http://doi.acm.org/10.1145/3020078.3021742.

[110] Raymond Tay. *OpenCL Parallel Programming Development Cookbook*. Sept. 2013. ISBN: 1849694524.

[111] Donald E. Thomas, Elizabeth D. Lagnese, John A. Nestor, et al. *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*. USA: Kluwer Academic Publishers, 1989. ISBN: 0792390539.

[112] *Top500 List - November 2019*. 2019. URL: www.top500.org/lists/2019/11/.

[113] A. Vaishnav, K. Pham, and D. Koch. "Live Migration for OpenCL FPGA Accelerators". In: *2018 International Conference on Field-Programmable Technology (FPT)*. 2018, pp. 38–45. DOI: 10.1109/FPT.2018.00017.

[114] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. "A Survey on FPGA Virtualization". In: *28th International Conference on Field Programmable Logic and Applications, FPL18, Dublin, Ireland, September 2018*. 2018.

[115] C. Vatsolakis and D. Pnevmatikatos. "RACOS: Transparent access and virtualization of reconfigurable hardware accelerators". In: *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2017, pp. 11–19. DOI: 10.1109/SAMOS.2017.8344606.

[116] Malte Vesper, Dirk Koch, Kizheppatt Vipin, et al. "JetStream: An open-source high-performance PCI Express 3 streaming library for FPGA-to-Host and FPGA-to-FPGA communication". In: *26th International Conference on Field Programmable Logic and Applications, FPL 2016, Lausanne, Switzerland, August 29 - September 2, 2016*. 2016, pp. 1–9. DOI: 10.1109/FPL.2016.7577334. URL: https://doi.org/10.1109/FPL.2016.7577334.

[117] J. Vienne, J. Chen, M. Wasi-Ur-Rahman, et al. "Performance Analysis and Evaluation of InfiniBand FDR and 40GigE RoCE on HPC and Cloud Computing Systems". In: *2012 IEEE 20th Annual Symposium on High-Performance Interconnects*. 2012, pp. 48–55. DOI: 10.1109/HOTI.2012.19.

[118] Venkatasubramanian Viswanathan, Rabie Ben Atitallah, Jean-Luc Dekeyser, et al. "A Parallel And Scalable Multi-FPGA based Architecture for High Performance Applications (Abstract Only)". In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 22-24, 2015*. 2015, p. 266. DOI: 10.1145/2684746.2689115. URL: https://doi.org/10.1145/2684746.2689115.

[119] *Vivado Design Suite User Guide v2017.4*. 2017. URL: www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug893-vivado-ide.pdf.

[120] D. V. Vu, O. Sander, T. Sandmann, et al. "Enabling partial reconfiguration for coprocessors in mixed criticality multicore systems using PCI express single-root I/O virtualization". In: *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*. 2014, pp. 1–6. DOI: 10.1109/ReConFig.2014.7032516.

[121] Ze-ke Wang, Shuhao Zhang, Bingsheng He, et al. "Melia: A MapReduce Framework on OpenCL-Based FPGAs". In: *IEEE Trans. Parallel Distrib. Syst.* 27.12 (2016), pp. 3547–3560. DOI: 10.1109/TPDS.2016.2537805. URL: https://doi.org/10.1109/TPDS.2016.2537805.

[122] Jagath Weerasinghe, Raphael Polig, François Abel, et al. "Network-attached FPGAs for data center applications". In: *2016 International Conference on Field-Programmable Technology (FPT)*. 2016, pp. 36–43. DOI: 10.1109/FPT.2016.7929186.

[123] Dennis Weller, Fabian Oboril, Dimitar Lukarski, et al. "Energy Efficient Scientific Computing on FPGAs Using OpenCL". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. New York, NY, USA: Association for Computing Machinery, 2017, 247–256. ISBN: 9781450343541. DOI: 10.1145/3020078.3021730. URL: https://doi.org/10.1145/3020078.3021730.

[124] C. Whitson and M. Michelsen. "The negative flash". In: *Fluid Phase Equilibria, vol. 35*. 1989, pp. 51–71.

[125] Masato Yoshimi, Yuri Nishikawa, Mitsunori Miki, et al. "A Performance Evaluation of CUBE: One-Dimensional 512 FPGA Cluster". In: *Reconfigurable Computing: Architectures, Tools and Applications, 6th International Symposium, ARC 2010, Bangkok, Thailand, March 17-19, 2010. Proceedings*. 2010, pp. 372–381. DOI: 10.1007/978-3-642-12133-3_36. URL: https://doi.org/10.1007/978-3-642-12133-3_36.

[126] *Zynq UltraScale+ Device Technical Reference Manual*. 2018. URL: www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf.