

POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

OPTIMIZATION OF FPGA-BASED HETEROGENEOUS MULTI-CORE SYSTEMS-ON-CHIP FOR DATA CENTERS

Doctoral Dissertation of:
Gabriele Montanaro

Supervisor:
Prof. Davide Zoni

Tutor:
Prof. Francesco Amigoni

The Chair of the Doctoral Program:
Prof. Luigi Piroddi

Academic Year 2024/2025 – Cycle XXXVII

Ai miei genitori Susanna ed Eugenio,
per avermi sostenuto in ogni modo possibile.

Alla mia ragazza Sara, per l'incondizionata
fiducia che ha sempre riposto in me.

Acknowledgments

This PhD has been a long journey, marked by exciting achievements and, at times, a few difficult moments. It is the kind of experience you can truly understand only once you have gone through it: nothing can really prepare you for it. And I think that is part of what makes it so interesting. There is no guide on how to do a PhD. There is no single right or wrong way. Each path is unique, and you have to think carefully to navigate the many challenges you will encounter along the way. Inevitably, you will make mistakes, and often you will have to learn things the hard way. But it is only at the end of the journey that you can look back and fully appreciate the value of each step, how much you have learned, and how deeply your approach to study and research has evolved.

Even if every PhD is unique, it is not something that can be done alone, and I am deeply grateful to all those who shared, in different ways, parts of this journey with me. My biggest thanks go to my advisor, Davide: it goes without saying that this research would not have been possible without his support. Whenever I was stuck, he always managed to come up with the right idea or a new perspective, allowing me to move forward with my work. Another special thanks goes to my colleague Andrea Galimberti: as the most senior PhD student in the lab, he has been a mentor to me, and he contributed significantly to the writing of many of the papers we published together. If I have learned how to write, structure, and publish an article (and this thesis as well), much of the credit goes to him. I also want to acknowledge Professor Francesco Trovò for his contribution to the online-learning SoC optimization methodology: without his expertise in machine learning, it would not have

been possible to develop such an effective approach. Finally, I would like to thank all the colleagues who, in one way or another, shared part of this journey with me: Davide Galli, Michele, Giuseppe, Andrea Motta, Adriano, and all the graduate and undergraduate students who spent time in Office 4 of Building 21 over the years.

Of course, this is also a journey I lived outside the office. I want to thank my parents, Susanna and Eugenio, for always encouraging me to do my best. I am especially grateful to my girlfriend, Sara, who patiently listened to all my concerns and offered thoughtful advice, even without being familiar with the details of my research. Most of all, she has always had great faith in me as a researcher, and that gave me more strength than she probably realizes. Finally, I want to thank all my friends, far too many to name individually. I have not always managed to explain what my research is about, but I still received plenty of advice anyway—and, just as importantly, shared many great moments with them.

Abstract

In recent years, the demand for computational power–especially from artificial intelligence applications–has grown at an unprecedented pace. To address this need, massive investments have been made in the construction of data centers and high-performance computing systems. However, traditional architectures such as CPUs and GPUs have shown clear limitations: their versatility comes at the cost of low efficiency in domain-specific contexts, both in terms of performance and energy consumption. This has led to the adoption of highly specialized hardware accelerators, capable of improving computational performance while simultaneously reducing energy costs.

Among the available solutions, FPGAs stand out thanks to their ability to implement architectures tailored to specific operations, combined with the flexibility of rapid reprogramming, which enables much shorter development cycles compared to ASICs. Nevertheless, their cost and non-negligible energy consumption make optimized usage essential. Furthermore, the ever-increasing size of modern FPGAs allows them to host multiple applications simultaneously, but also dramatically enlarges the design space of possible configurations. Exhaustively testing all configurations becomes infeasible, which has motivated the development of design space exploration approaches to accelerate the identification of optimal solutions.

In this context, the research presented in this thesis introduces a set of architectural and methodological techniques to facilitate design space exploration in systems with multiple hardware accelerators instantiated on a single FPGA. The work is structured into two main directions.

The first concerns the development of a prototyping platform for heterogeneous systems, enhanced with features designed to: *(i)* expand the design space by balancing energy consumption, resource utilization, and computational performance; *(ii)* collect execution data from the explored configurations; and *(iii)* accelerate the exploration process through the use of FPGAs' dynamic partial reconfiguration. Experimental results demonstrate that these techniques reduce the design space exploration time by up to 29×.

The second research direction focuses on optimization methodologies, pursued along two complementary levels: at the application level, by identifying which sections should run in software and which in hardware; and at the system level, by determining the best allocation of accelerators to maximize performance while minimizing area usage. In the latter case, the use of machine learning techniques makes the methodology architecture-agnostic while drastically reducing the number of configurations that need to be evaluated. Results show that the optimal configuration can be identified by exploring less than 0.1% of the overall design space.

Sommario

Negli ultimi anni la domanda di potenza computazionale, in particolare da parte delle applicazioni basate sull'intelligenza artificiale, è cresciuta ad un ritmo senza precedenti. Per rispondere a questa esigenza si è assistito a ingenti investimenti nella costruzione di data center e sistemi di calcolo ad alte prestazioni. Tuttavia, le architetture tradizionali, come CPU e GPU, risultano limitate: la loro versatilità comporta infatti una bassa efficienza nell'esecuzione di applicazioni ad elevata specificità, sia in termini di prestazioni che di consumo energetico. Questo ha portato all'adozione di acceleratori hardware altamente specializzati, capaci di migliorare le performance riducendo allo stesso tempo i costi energetici.

Tra le diverse soluzioni, gli FPGA si distinguono per la possibilità di implementare architetture dedicate ad operazioni mirate e di essere riprogrammati rapidamente, offrendo cicli di sviluppo molto più brevi rispetto agli ASIC. Nonostante ciò, il loro costo e il consumo energetico non trascurabile rendono cruciale un'ottimizzazione del loro utilizzo. Inoltre, le dimensioni sempre maggiori degli FPGA di ultima generazione permettono di ospitare più applicazioni contemporaneamente, ma ampliano enormemente lo spazio delle possibili configurazioni. Testarle tutte diventa impraticabile, motivo per cui sono stati sviluppati approcci per l'esplorazione dello spazio di design in grado di ridurre i tempi necessari per identificare le configurazioni ottimali.

In questo contesto, la ricerca presentata in questa tesi introduce una serie di tecniche architetturali e metodologiche per facilitare l'esplorazione dello

spazio di design in sistemi con più acceleratori hardware istanziati su un singolo FPGA. Il lavoro si articola in due direzioni principali.

La prima riguarda lo sviluppo di una piattaforma di prototipazione per sistemi eterogenei, arricchita con funzionalità pensate per: *(i)* ampliare lo spazio di design, bilanciando consumo energetico, utilizzo di risorse e capacità computazionale; *(ii)* raccogliere dati di esecuzione sulle configurazioni esplorate; *(iii)* accelerare il processo di esplorazione grazie alla riconfigurabilità parziale dinamica degli FPGA. Gli esperimenti condotti mostrano come tali tecniche consentano una riduzione fino a 29 volte del tempo di esplorazione.

La seconda direzione di ricerca si concentra invece su metodologie di ottimizzazione, perseguite su due livelli complementari: quello dell'applicazione, stabilendo quali sezioni eseguire in software e quali in hardware, e quello di sistema, individuando la migliore allocazione degli acceleratori per massimizzare le prestazioni minimizzando l'area occupata. In quest'ultimo caso, l'uso di tecniche di machine learning permette di rendere la metodologia indipendente dall'architettura specifica, e al contempo ridurre drasticamente il numero di configurazioni da valutare. I risultati dimostrano che è possibile individuare la configurazione ottimale esplorando meno dello 0.1% dello spazio di design complessivo.

Contents

1 Introduction **1**
- 1.1 Contributions . . . 8
  - 1.1.1 Goal of the thesis . . . 9
  - 1.1.2 Output intellectual properties . . . 11
- 1.2 Structure of the thesis . . . 12

2 Background **13**
- 2.1 SoC design . . . 13
  - 2.1.1 NoC . . . 14
  - 2.1.2 Frequency and voltage islands . . . 15
  - 2.1.3 High-level synthesis . . . 15
  - 2.1.4 AXI protocol . . . 17
- 2.2 SoC optimization . . . 19
- 2.3 Dynamic partial reconfiguration . . . 20

3 Related works **23**
- 3.1 SoC prototyping frameworks . . . 24
  - 3.1.1 ESP . . . 25
  - 3.1.2 Chipyard . . . 27
  - 3.1.3 PULP . . . 28
- 3.2 SoC optimization . . . 28
  - 3.2.1 HW/SW co-design . . . 29
  - 3.2.2 Accelerator mapping . . . 30
- 3.3 Dynamic voltage and frequency scaling . . . 31

3.4 Dynamic partial reconfiguration 32
3.5 Memory benchmarking platforms 33

4 Omega SoC prototyping platform **35**
4.1 Architecture overview 37
4.2 Configurable-DFS frequency-island architecture 38
4.3 Multi-replica accelerator-tile architecture 40
4.4 DPR mechanism 41
4.5 Memory benchmarking 43
4.6 Run-time monitoring infrastructure 46
4.7 Full DSE flow 47

5 Optimization policies **51**
5.1 Accelerator-level optimization 52
5.1.1 Performance profiling 52
5.1.2 High-level synthesis of hardware accelerators 53
5.1.3 Integration of hardware accelerators in heterogeneous SoC 53
5.2 FARMER: an online-learning optimization methodology . . 53
5.2.1 Optimization problem formulation 55
5.2.2 ML-based regressor 56
5.2.3 Online decision rule 57
5.2.4 In-circuit exploration flow 58

6 Experimental results **63**
6.1 Omega SoC characterization 64
6.1.1 Experimental setup 66
6.1.2 Multi-replica accelerator tiles: area-performance analysis 69
6.1.3 Configurable-DFS frequency islands: area analysis . . 69
6.1.4 SoC accelerators: performance analysis 71
6.1.5 Memory: traffic analysis 73
6.2 Memory benchmarking 73
6.2.1 DDR SDRAM background 74
6.2.2 Experimental setup 76
6.2.3 Throughput results 77
6.2.4 Performance breakdown of mixed workloads 78
6.2.5 Performance analysis 78
6.3 Design space exploration with DPR 80
6.3.1 Experimental setup 80
6.3.2 Experimental results 83

6.4 HW/SW co-design results 86
6.4.1 BIKE HW/SW co-design 86
6.4.2 Experimental setup 89
6.4.3 Area results 89
6.4.4 Performance results 90
6.5 FARMER evaluation 92
6.5.1 Experimental setup 92
6.5.2 Exhaustive simulation 97
6.5.3 Realistic scenarios 98
6.6 DPR-accelerated DSE 99
6.6.1 Experimental setup 100
6.6.2 Exhaustive scenario 102
6.6.3 Realistic scenarios 105

7 Conclusions **109**

Appendices **111**

A List of publications **113**

Bibliography **121**

CHAPTER 1

Introduction

Data centers have become fundamental infrastructures in modern society, supporting a wide range of applications, from social media, gaming, and web browsing to scientific computing and the training of artificial intelligence (AI) models. Hyperscale computing facilities are now ubiquitous, shaping the daily experiences of billions of individuals. The significance of this domain continues to expand, with the global market valued at $372.8 billion USD in 2023 and projected to grow to $624 billion USD by 2029 [3]. Traditionally, central processing units (CPUs) have served as the primary computational units in data centers, excelling in sequential processing and handling diverse workloads. However, the increasing prevalence of specialized tasks, particularly in AI and machine learning, has driven the widespread adoption of graphics processing units (GPUs), which offer superior performance in parallel computations [70]. Despite the growing number of GPUs, particularly in new facilities designed for large-scale AI model training, CPUs remain indispensable due to their efficiency in executing single-threaded and general-purpose workloads.

Despite their critical role, data centers face several emerging challenges. First, the continuous demand for higher computational performance can no longer be sustained solely by CPUs, given the stagnation of Moore's

Law [76] and the breakdown of Dennard scaling [33]. Although GPUs provide substantial parallelism, their architectural characteristics make them unsuitable for certain applications, particularly those exhibiting highly irregular computational patterns [70]. Second, the escalating power consumption of hyperscale computing facilities has become increasingly unsustainable due to rising energy costs, the growing complexity and expense of cooling infrastructures [115], and heightened awareness of the environmental impact associated with such extensive resource utilization [95].

In this context, field-programmable gate arrays (FPGAs) have emerged as a promising technology to address the challenges of performance and energy efficiency in modern computing systems. FPGAs offer the ability to reconfigure their logic to implement diverse computational kernels, providing a significant advantage over other existing architectures. Compared to CPUs, FPGAs are more specialized for specific computational tasks, enabling higher energy efficiency and greater parallelism. In contrast to GPUs, FPGAs demonstrate superior performance in various application domains, such as high-frequency trading [53], smart networking interfaces [106], and genomic mapping [28], due to their low latency, high reconfigurability, and suitability for irregular computations. Furthermore, FPGA reconfigurability offers a key advantage over application-specific integrated circuits (ASICs), as the dynamic and rapidly evolving workloads in data centers necessitate shorter time-to-market cycles and the ability to adapt applications without requiring new hardware deployments.

The adoption of FPGAs in hyperscale computing facilities has significantly accelerated over the past decade. This growth is primarily driven by the decline of Moore's Law and the increasing necessity for hardware acceleration to enhance computational performance. Additionally, this trend has been reinforced by several key technological advancements. The introduction of silicon interposer technology has enabled a substantial increase in FPGA capacity. For instance, the AMD Virtex-7 2000T, released in 2011, was the first FPGA to surpass one million logic elements [1], whereas the most advanced FPGA currently available, the AMD Versal VP1902, features 18.5 million logic elements [7]. This dramatic expansion in logic capacity, whose timeline is depicted in Figure 1.1, allows FPGAs to implement a wide range of accelerators with a high degree of parallelism. Furthermore, FPGA manufacturers have integrated high-performance memory solutions, including both off-chip double-data rate (DDR) memory and on-chip high-bandwidth memory (HBM), along with high-speed interconnect technologies such as PCIe. These advancements enable modern FPGA platforms

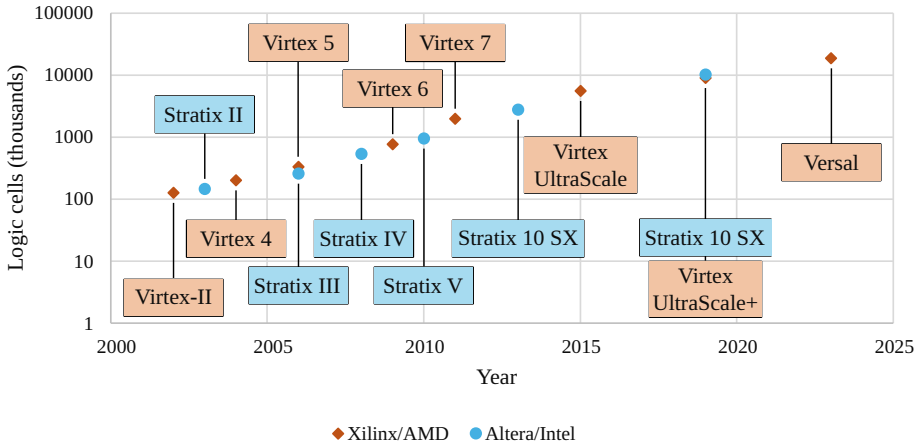


Figure 1.1: *Comparison of the largest devices for each FPGA generation from the two major vendors during the last 25 years. Size is measured in terms of logic cells (LC) for Xilinx/AMD FPGAs and logic elements (LE) for Altera/Intel FPGAs.*

to efficiently manage large-scale data processing while maintaining high throughput and low latency.

One of the earliest successful deployments of FPGAs in data centers dates back to 2015, when Microsoft introduced the Catapult infrastructure to accelerate web searches within the Bing search engine [27]. This adoption led to a remarkable 95% improvement in performance with only a 10% increase in power consumption. The following year, Amazon Web Services (AWS) launched the F1 instances [6], featuring eight AMD Virtex UltraScale+ FPGAs per instance, providing users with access to FPGA-based acceleration for their computations. Similarly, in 2017, other important players like Alibaba [4], Tencent [8], Baidu [2], and Huawei [5], introduced FPGA-as-a-Service (FaaS) offerings, enabling customers to leverage FPGA hardware acceleration for various applications. While most major cloud service providers introduced FPGAs into their infrastructures between 2016 and 2017, they have continued upgrading their data centers with newer generations of devices Two examples are AWS EC2 F2 instances [21] and Microsoft Azure NP-series virtual machines [9], both launched in 2024.

As exemplified in Figure 1.2, that reports major milestones in FPGAs adoption in the cloud, these devices have not only been commercialized as FaaS offerings but have also been adopted to accelerate specific workloads. A prominent example is the DRAGEN Bio-IT platform, which leverages AWS F1 instances to accelerate genomic sequencing pipelines [65]. Another

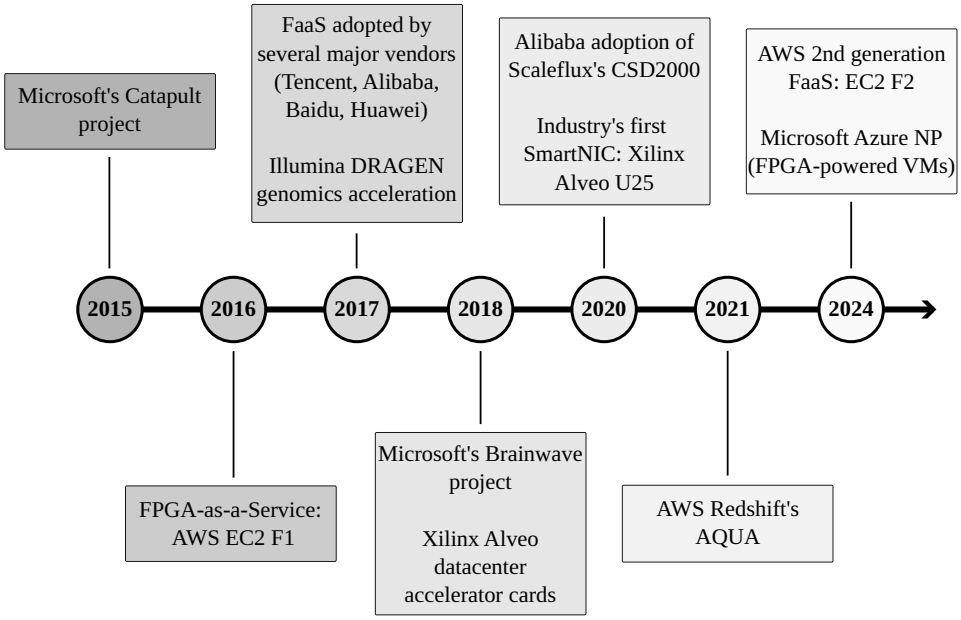


Figure 1.2: *A timeline with some of the most important milestones in the adoption of FPGAs in industrial data centers.*

notable project is Microsoft's Brainwave, designed to accelerate deep neural network inference by exploiting the inherent parallelism and low latency of FPGAs. When first introduced in 2018, Brainwave achieved performance gains exceeding an order of magnitude compared to state-of-the-art GPUs [34]. Another relevant example is the adoption of second-generation ScaleFlux Computational Storage Drives (CSDs) in Alibaba data centers. These devices employ near-memory FPGAs to accelerate frequently executed tasks, such as compression and decompression, on large data batches, thereby offloading computational workload from CPUs and reducing congestion in interconnects. Similarly, AWS AQUA (Advanced Query Accelerator) employs FPGAs to accelerate data queries within AWS data centers, delivering up to 10× performance improvements [20].

Despite these advancements, the integration of FPGAs in data centers still presents significant challenges. The primary obstacles stem from the complexity of the hardware design process, which demands specialized expertise and results in longer time-to-market compared to software-based solutions. Additionally, FPGA design involves a significantly larger design space than software-only approaches, necessitating the use of sophisticated optimization techniques to maximize performance and efficiently manage resource utilization.

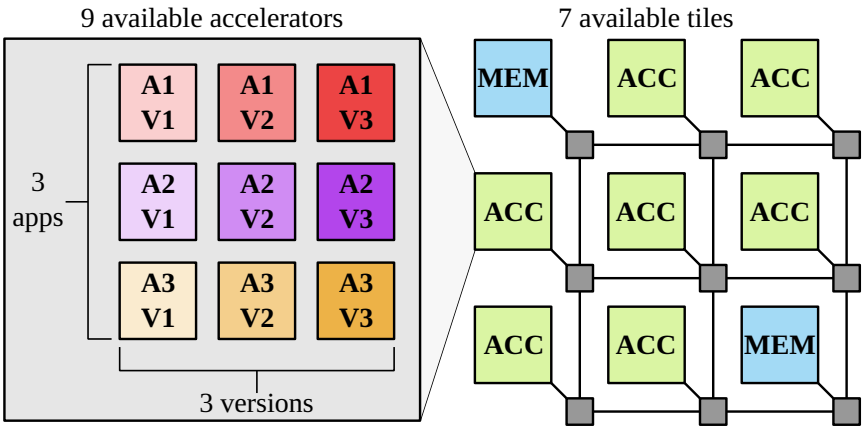


Figure 1.3: *Design space exploration example for a 3×3 NoC implementing different accelerators.*

For these reasons, substantial research efforts have been dedicated to the efficient deployment of FPGAs in data centers. A particularly noteworthy area of investigation focuses on the development of multi-tenancy frameworks for FPGA-based infrastructures [23]. As previously discussed, modern high-performance computing (HPC) FPGAs feature millions of logic elements, often exceeding the resource requirements of a single application. However, these devices are costly, and their underutilization represents a significant inefficiency for service providers. A viable solution to this issue is the implementation of FPGA multi-tenancy, which enables the concurrent execution of multiple independent applications by partitioning and sharing the device's logic resources among different users. This approach maximizes FPGA utilization and improves cost efficiency. To achieve multi-tenancy, virtualization techniques have been introduced, allowing each user's tasks to be transparently mapped to a designated portion of the FPGA. Resources are allocated and deallocated dynamically as needed. In this context, dynamic partial reconfiguration (DPR) plays a critical role by enabling the reconfiguration of specific regions of the device without disrupting the execution of other users' applications. Several multi-tenancy frameworks have been proposed in the literature, focusing on different aspects such as resource efficiency [56], high-speed connectivity [69], and the development of FPGA-specific operating systems [60]. However, the primary limitation hindering the widespread adoption of these frameworks in real-world deployments is their security vulnerabilities. Malicious users can exploit direct access to the FPGA's physical layer to launch various types of attacks, potentially compromising the server infrastructure or other users' workloads [23].

Beyond the efficient utilization of FPGA resources, other challenges associated with FPGA adoption in data centers have also been extensively investigated in the literature. One critical area of research focuses on improving the productivity of the FPGA design process by abstracting low-level hardware details from end users, promoting design and intellectual property (IP) reuse, and reducing time-to-market. In this context, system-on-chip (SoC) prototyping platforms [13, 68] offer a compelling solution, as they provide integrated methodologies for the rapid development of multi-core, heterogeneous SoCs. Given the close relevance of these frameworks to the topics covered in this thesis, a detailed analysis of the main approaches in this domain will be presented in Chapter 3. Other research efforts have instead focused on the design of FPGA-based accelerators for server applications [28, 53, 106], demonstrating the effectiveness of hardware acceleration in data center environments.

Given the cost and energy consumption associated with FPGA deployment in data centers, optimizing a design with respect to its power, performance, and area (PPA) characteristics is of paramount importance. This optimization is typically achieved by adjusting various architectural and performance parameters, such as the degree of parallelization in accelerator computing elements, the decision to execute workloads in software versus implementing them in hardware, the operating frequency, and the size and topology of interconnects. However, the inherent complexity of HPC-oriented, heterogeneous FPGA architectures renders the identification of optimal configurations impractical, even for relatively small designs.

As an illustrative example, consider the small-scale scenario depicted in Figure 1.3. A 3×3 network-on-chip (NoC) with a 2D-mesh topology interconnects nine tiles, two of which are dedicated to memory connections (`MEM`), while the remaining seven (`ACC`) can each host a single accelerator. The target system is required to implement three applications - `A1`, `A2`, and `A3` - with each application having three distinct accelerator implementations (`V1`, `V2`, and `V3`), each offering different PPA trade-offs. Consequently, each `ACC` tile has nine possible configurations. Given that the system comprises seven `ACC` tiles, the total number of possible design configurations amounts to 9^7, which is approximately 4.8 millions.

The exploration of such an extensive design space cannot be conducted purely algorithmically, as the complex interactions among computing elements that concurrently access the same interconnect introduce significant variability. Figure 1.4 illustrates the estimated and actual performance of a hardware-implemented *dfadd* accelerator, adapted from [40], executing on an AMD Virtex-7 2000T FPGA. The accelerator operates within a 2D-mesh

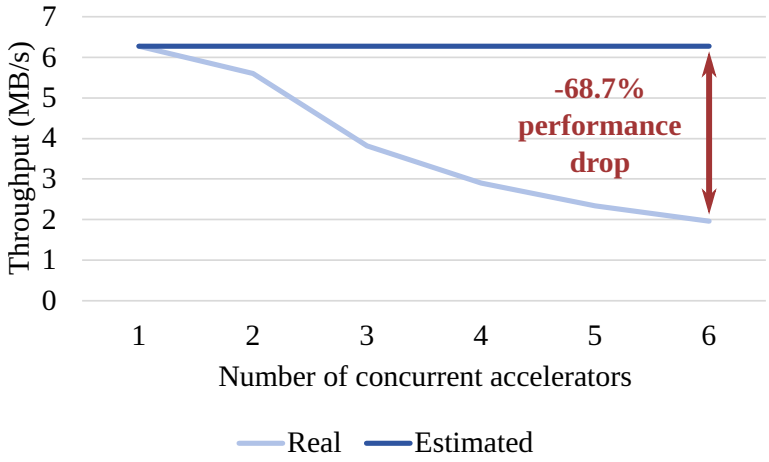


Figure 1.4: *Real and estimated throughput of a dfadd accelerator with different numbers of accelerators sharing the same memory.*

architecture featuring a single shared memory, with a variable number of additional hardware accelerators competing for memory access. A naive performance estimation assumes a constant throughput regardless of contention; however, empirical results demonstrate a degradation in performance as memory contention increases. This discrepancy underscores the inherent complexity of multi-accelerator architectures, where interactions among concurrently executing hardware accelerators cannot be easily modeled or generalized.

Current solutions for evaluating system performance typically require comprehensive testing to extract execution and performance statistics. These approaches can generally be categorized into two main strategies. The first is the simulation-based approach, which utilizes cycle-accurate simulators [22] to model the system and derive the necessary performance metrics. However, simulating large systems can be time-consuming, often requiring several hours or even days. Additionally, the accuracy of such simulations is highly dependent on the fidelity of the models used to represent the hardware components of the design. Alternatively, the system can be prototyped by directly implementing it on an FPGA device. This method significantly reduces execution time and improves the accuracy of performance data collection. However, the implementation phase remains time-intensive, with complex designs still requiring several hours to be fully realized on hardware.

1.1 Contributions

FPGAs are playing an increasingly pivotal role in data centers, accelerating the execution of various applications while simultaneously reducing power consumption. In this context, identifying the optimal configuration of the heterogeneous architecture instantiated on the FPGA is crucial for enhancing the PPA characteristics of the device. Such optimization can lead to improved data center performance, alongside reductions in energy consumption and equipment costs. However, the vast design space of large heterogeneous SoCs, coupled with the inherent complexity of hardware design, makes achieving this objective particularly challenging.

This thesis presents a novel framework for the design space exploration (DSE) of complex heterogeneous SoCs, facilitating the identification of optimal configurations. The framework addresses various aspects of the problem, including the low-level hardware features that enable the rapid setup and testing of different SoC configurations, as well as the development of an automated flow for design space exploration. This research makes three primary contributions.

Novel DSE-oriented SoC prototyping platform This thesis introduces the first DSE-oriented SoC prototyping platform. While several frameworks in the literature facilitate the rapid generation, simulation, and synthesis of parametric SoCs, none specifically focus on the automatic exploration of the design space. Notably, the proposed framework incorporates several key features: a dynamic frequency scaling (DFS) infrastructure that enables the adjustment of clock domain frequencies at runtime; a multi-replica accelerator-tile architecture, which allows the designer to select the parallelization of hardware accelerators at design time; a memory benchmarking platform to evaluate the memory performance in a range of traffic conditions; and a DPR infrastructure that permits the reconfiguration of deployed accelerators during runtime.

Application- and system-level SoC optimization methodologies This research explores the methodological aspects involved in identifying the optimal solution, aiming to overcome the limitations of the narrow-scoped approaches typically found in state-of-the-art heuristics. The goal is to adopt strategies that can be broadly generalized across diverse systems and architectures. This is accomplished at both the application level, by determining which components of an application should be executed in hardware versus

software, and at the system level, by selecting the optimal allocation of accelerators on the SoC using an online learning methodology.

Complete experimental flow This research presents a comprehensive experimental flow that enables the automatic exploration of the design space by evaluating multiple configurations of the SoC architecture on a real FPGA device to identify the optimal configuration. This is achieved by leveraging the DPR capabilities of the framework, which facilitates the rapid instantiation of new configurations without the need to implement each one from scratch. The exploration process can be conducted either exhaustively or by employing the previously mentioned online-learning methodology to reduce the number of configurations tested.

1.1.1 Goal of the thesis

Ultimately, the goal of this thesis is to provide a hardware infrastructure and optimization methodologies that allow the rapid exploration of the design space of complex, heterogeneous SoCs. This is represented in Figure 1.5, which summarizes the full DSE flow proposed in this work.

Starting from a set of features characterizing the desired SoC, like the set of applications, some requirements on the throughput, a set of rules for the target SoC (e.g., the network-on-chip topology or the allowed parallelism levels for an accelerator), and the resource constraints of the device, this flow can automatically identify the optimal configuration of the SoC, i.e., the configuration that satisfies all the characteristics requested for the SoC and that maximizes a certain optimization function. The flow can be divided into two main stages: an initialization stage, which is executed only once, and the iterative pipeline, which includes the machine learning (ML)-based configuration selection and the on-board testing, and it is repeated for a predefined number of times.

The initialization stage starts with the `Synth & impl.` phase, which consists of the synthesis, implementation, and bitstream generation, both for the static and the dynamic regions of the FPGA. This is the most computationally-intensive part of the whole flow, but with the use of dynamic partial reconfiguration it can be brought outside the iterative loop enabling a reduction in exploration time with respect to traditional approaches by more than an order of magnitude, as an implementation could require tens of minutes while flashing partial bitstreams needs just a few seconds. Once the `Synth & impl.` phase is completed, the static bitstream, containing the SoC with empty tiles, can be flashed onto the FPGA (`SoC flash`). At

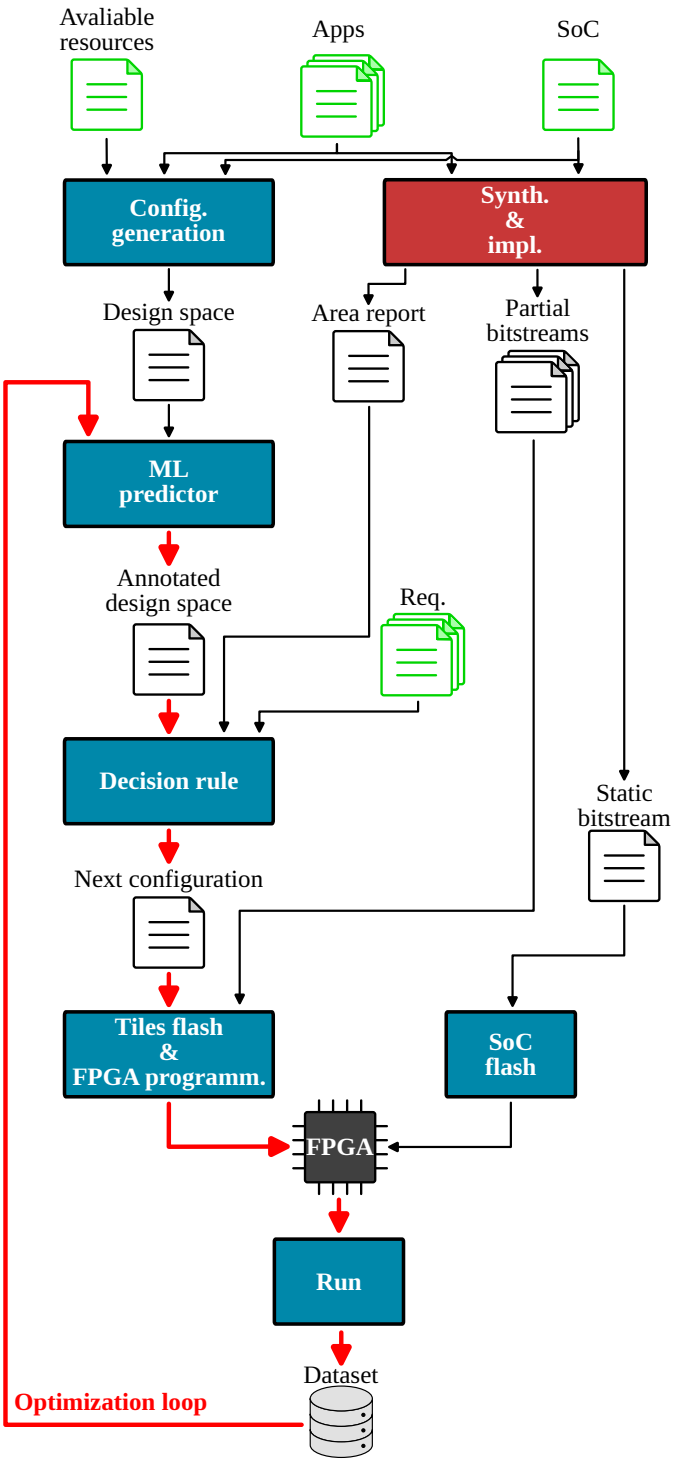


Figure 1.5: *An overview of the DSE flow proposed in this thesis.*

the same time, the framework employs the information regarding the SoC rules, the selected applications, and the available resources to generate a list containing all the possible configurations of the SoC, i.e., the design space.

The `ML predictor` block is the entry point of the feedback loop, receiving the list of configurations as well as the dataset containing the statistics related to the configurations that have already been tested by the flow. The decision rule relies on such estimates to select the following configuration to be tested (`Next configuration`), trying to choose a configuration that maximizes the obtained information and which is feasible according to the requirements and the optimization goal.

At this point, the partial bitstreams corresponding to the accelerators included in the selected configuration are individually flashed onto the device, and then the testbench executable is loaded onto the SoC (`Tiles flash & FPGA programm.`). The program can run and collect the execution statistics, which are sent back to the host PC to be added to the dataset. For each tested configuration, the dataset grows, making the estimates of the ML predictors more accurate.

At the end of the iterative procedure, this flow returns the best configuration among those available in the dataset, i.e., the configuration that satisfies all the requirements and that maximizes the optimization function.

1.1.2 Output intellectual properties

The practical and engineering focus of this thesis has led to the development of two IPs: the DSE-oriented SoC prototyping platform Omega and the online-learning-based SoC optimization framework FARMER.

Omega Omega constitutes the hardware-software infrastructure at the core of this research. It is a SoC prototyping platform built upon the open-source ESP framework [68], extended with several novel features described in Chapter 4. The primary contribution of Omega lies in its emphasis on design space exploration. Its components are specifically designed to maximize the configurability of the generated SoC, enabling fine-grained tuning of the system's PPA characteristics, as well as to accelerate the exploration process through the use of DPR. Omega has been thoroughly described in a dedicated article [73], whose contents have been reported in this thesis.

FARMER FARMER is a novel online-learning methodology for allocating a set of hardware accelerators within an SoC instantiated on an FPGA. To the best of our knowledge, it represents the first SoC allocation strategy that

explicitly leverages machine learning to guide the exploration of the design space. The core mechanisms underlying FARMER are detailed in Chapter 5. FARMER has been presented in its original form in [74] and in an extended, DPR-enhanced version in [75].

1.2 Structure of the thesis

The remainder of this thesis is organized as follows. Chapter 2 provides the necessary background for the topics addressed in this research. In particular, it introduces concepts related to SoC design and optimization, as well as the fundamentals of dynamic partial reconfiguration technology. Chapter 3 reviews the state of the art on these topics, offering a comprehensive survey of the related literature. Chapter 4 presents the DSE-oriented SoC prototyping framework developed in this work, highlighting the features that set it apart from existing platforms in the state of the art. Chapter 5 describes the SoC optimization methodologies proposed here, namely the HW/SW co-design strategy for application-level optimization and the ML-based allocation algorithm for system-level optimization. Chapter 6 reports a wide range of experimental results, including the characterization of the SoC prototyping platform, the evaluation of the proposed optimization methodologies, and the integration of the complete DSE flow combining the framework with the optimization strategies. Finally, Chapter 7 summarizes the conclusions of this work.

CHAPTER 2

Background

This chapter introduces the fundamental concepts that form the basis of the research developed in this thesis. Given that the main contributions of this work concern the development of a novel, DSE-oriented SoC generator and methodologies for optimizing SoC designs, it is first necessary to clarify what a SoC is, along with the standard design and optimization practices applied to such architectures. In addition, a proper understanding of this thesis requires familiarity with DPR, an FPGA technology that plays a central role in the proposed SoC prototyping framework.

The chapter is structured as follows. Section 2.1 defines the concept of an SoC, its principal components, and its most common features. Section 2.2 outlines the key concepts related to SoC optimization. Finally, Section 2.3 introduces DPR technology and its associated terminology.

2.1 SoC design

A SoC integrates the major functional components of a computing system onto a single piece of silicon. Instead of relying on separate chips for the processor, memory, I/O controllers, and specialized accelerators, an SoC consolidates these elements into one integrated circuit. This integration

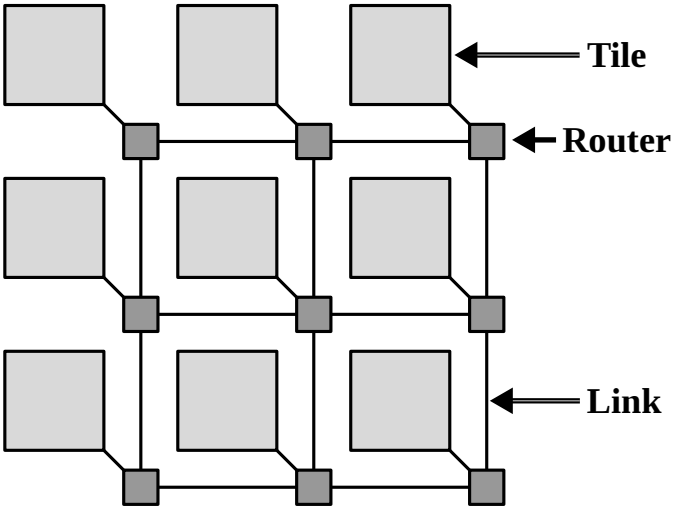


Figure 2.1: *An example of a 3×3 2D-mesh NoC.*

reduces physical footprint, power consumption, and communication latency, while enabling higher performance and greater energy efficiency. SoCs are now the dominant architecture in smartphones, embedded systems, and increasingly in data centers and high-performance computing.

The design of an SoC is inherently heterogeneous. A typical SoC may combine multiple general-purpose CPU cores, digital signal processors (DSPs), memory and input/output (I/O) controllers, and application-specific accelerators (e.g., for cryptography, machine learning, or multimedia). This heterogeneity introduces design complexity, particularly in terms of how these diverse components communicate and share resources.

2.1.1 NoC

As SoCs grew in scale, traditional shared buses or crossbar interconnects became insufficient due to bandwidth limitations, contention, and poor scalability. To address this, designers adopted NoCs–packet-switched communication fabrics that borrow principles from large-scale computer networks.

In a NoC, computation and memory components are organized into tiles, each tile typically containing a processing core (or cluster of cores), or an I/O controller, local cache, and a standardized interface towards the NoC. These tiles are connected through routers, which forward data packets across the chip in a scalable and structured manner. The network topology can vary, and each strategy has its own trade-offs in terms of latency, bandwidth, area,

and power. However, the most widespread NoC topology is the 2D-mesh, which is represented in Figure 2.1.

NoCs provide three critical benefits:

- **Scalability:** They support many-core systems without the bottlenecks of bus-based architectures.
- **Performance isolation:** Traffic from one component is less likely to block or starve others.
- **Modularity:** Designers can replicate and integrate tiles more easily, which simplifies floorplanning and verification.

2.1.2 Frequency and voltage islands

Another key concept in modern SoCs is the use of frequency and voltage islands. Instead of driving the entire chip with a single global clock and voltage domain, designers partition the SoC into regions (islands) that operate at independent frequencies and supply voltages. This partitioning provides two main advantages, power efficiency and performance tuning.

The former is attained by running low-activity components at reduced frequency or voltage, saving energy without affecting the performance of critical units. The latter is instead achieved by operating compute-intensive accelerators at higher clock speeds, while memory or peripheral units may remain slower but more energy-efficient.

The coexistence of multiple frequency and voltage islands requires sophisticated clock domain crossing (CDC) and power management strategies, but it is essential for balancing the competing goals of performance, energy efficiency, and thermal constraints in advanced SoCs.

2.1.3 High-level synthesis

The regular structure of NoCs enables systematic reuse of standardized components, since tiles can be replicated and composed in a modular fashion. This design style simplifies integration of heterogeneous processing and I/O units within tiles, as each can connect to the network through a uniform interface. The trend has been further accelerated by the adoption of high-level synthesis (HLS), which streamlines the generation of processing and peripheral blocks for different hardware architectures, making them easier to embed within NoC-based SoCs.

HLS is a hardware design methodology that is becoming increasing popular for fast prototyping of digital hardware designs. In HLS, circuits are

described in a high-level, software programming language (like C or C++). This code is then converted to a register-transfer level (RTL) description (VHDL or Verilog) with an HLS tool, that infers the expected behavior of the software code and generates an equivalent hardware design. This process is done with the help of pragmas and directives, employed by the designer to provide the tool with architectural details that cannot be directly described by the code, due to its time-agnostic and sequential nature.

This methodology presents some interesting advantages over the traditional RTL approach. First of all, the use of HLS brings a great improvement in productivity, allowing for a faster time-to-market. It is widely accepted that the design and verification of hardware circuits using RTL constitutes a bottleneck for productivity, since the designer needs to focus on the low-level details of the device. HLS instead allows to describe the application at a higher level of abstraction, leaving all the hardware complexity to the HLS tool and ultimately allowing to describe the same functionality with less lines of code.

The second, important benefit of using HLS is that it offers the possibility to make a design parametric with the use of generic pragmas. This directives can be used for a variety of purposes, from the unrolling of loops in order to increase the level of parallelism and reduce the latency, to the pipelining of a function to improve throughput and maximum clock frequency. The use of such pragmas leads to an expanded design space, that can be rapidly explored by changing the values assigned to those pragmas.

Finally, HLS helps bridging the gap between the software and hardware worlds. It allows software engineers to easily work on hardware designs without the need of learning VHDL or Verilog (though some hardware expertise is still needed to understand how the code will be mapped on hardware). It also allows to use the same code for hardware and software implementations of a design (with some limitations, since not all the software constructs can be translated to hardware), providing a precious methodology for HW/SW co-design. Furthermore, the testing and verification of software applications is much faster when compared to the hardware design flow, and applying the same approaches to the verification of hardware circuits can further improve the time-to-market of novel designs.

This advantages, however, come at the cost of a reduced quality of the design, in terms of performance, area, and timing. This is ultimately caused by the inherent lack of control over the resulting circuit experienced by the designer: since software code cannot be translated 1-to-1 to an RTL representation, the tool must infer some specifications of the design, and the choices made by the tool are not always the best ones (or the ones desired by

the designer given the specific use case). There is a lot of ongoing research with the goal of bridging this gap [62], but there is still much to do to increase the quality of results of HLS-generated designs to be comparable to RTL circuits.

Given all these considerations, the use of HLS is desirable when time-to-market and portability are more important than performance and area consumption. This is the case, for example, of many benchmarking suites, where the goal is usually to provide many circuits for a large number of target platforms [40]. However, when pushing a specific design to its limits is what matters the most, the traditional RTL flow is still the way to go.

2.1.4 AXI protocol

Standardized on-chip communication protocols are essential in NoC-based SoC design, as they provide well-defined interfaces that simplify integration of heterogeneous components. By decoupling the functional design of processing or I/O units from the underlying interconnect, standard protocols enable interoperability, reuse, and modularity. Among these, the Advanced eXtensible Interface (AXI) protocol–part of the ARM AMBA specification–has become the most widely adopted for tile-to-NoC interfaces due to its flexibility, performance, and ecosystem support.

The AXI protocol family defines three main variants, each optimized for different use cases:

- **AXI Full**: The most complete version, supporting burst-based transfers, multiple outstanding transactions, and separate read/write channels. It is designed for high-performance components such as CPUs, GPUs, and DMA engines, where throughput and low latency are critical.

- **AXI Lite**: A lightweight subset of the full protocol that removes burst capabilities and reduces complexity. AXI Lite is commonly used for control registers and low-bandwidth peripherals, where the simplicity of the interface outweighs the need for maximum performance.

- **AXI Stream**: A unidirectional, high-throughput streaming interface tailored for dataflow applications. Unlike AXI Full and Lite, it does not use address-based transactions; instead, it supports continuous streams of data, making it ideal for signal processing, multimedia, and machine learning accelerators.

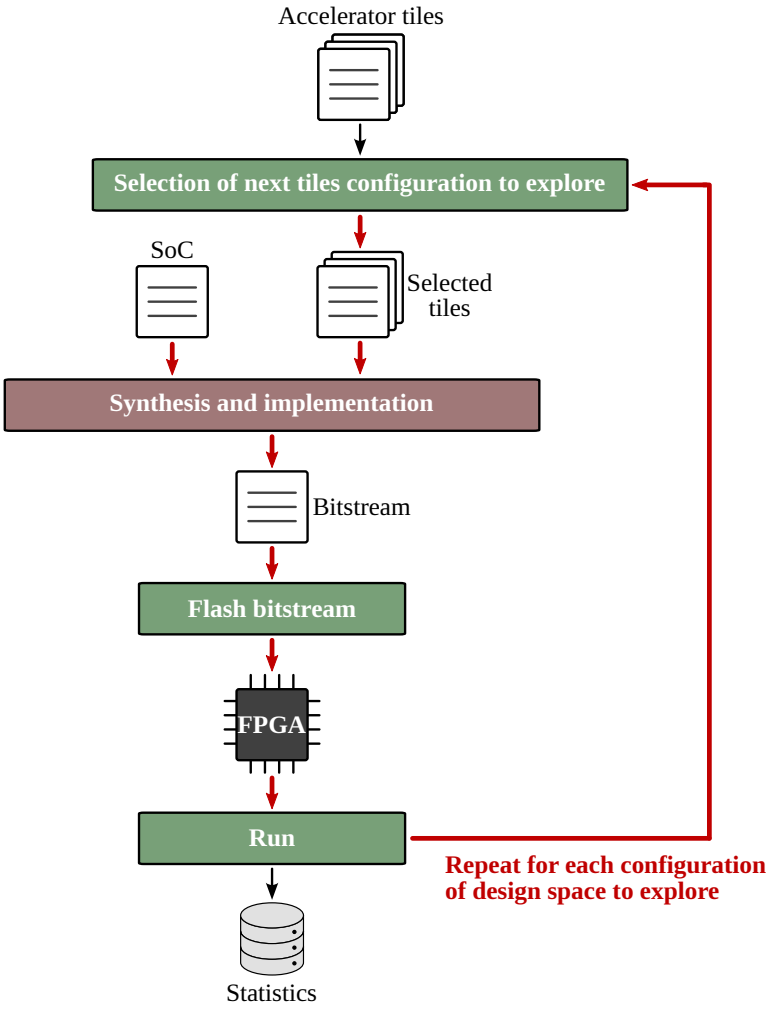


Figure 2.2: *High-level flow for the traditional DSE of FPGA-based heterogeneous multi-core SoCs.*

2.2 SoC optimization

Once the functional blocks to be integrated into the SoC have been identified, the next challenge is optimization. This can be approached at two distinct but related levels.

The first is intra-application optimization, which in the context of SoC design can be declined to the form of HW/SW co-design. Here the designer decides which portions of an application should be mapped to hardware accelerators and which should remain in software. This choice leads to several trade-offs, in terms of *(i)* performance, as hardware accelerators offer superior throughput with respect to software execution, *(ii)* area consumption, *(iii)* energy efficiency, and *(iv)* flexibility, since hardware accelerators are dedicated to a single task while a CPU can execute several different applications.

The second aspect is inter-application optimization, which concerns the integration of multiple accelerators and shared resources across the chip. Given the complexity of the interactions among heterogeneous computing elements, the designer must make several architectural decisions to meet performance, area, and power consumption requirements. First, the placement of the various tiles must account for the distance between computing elements and memory, as well as for potential congestion along interconnection paths. This is particularly critical for memory-bound applications, whose performance strongly depends on communication bandwidth and latency. Second, achieving the optimal balance between performance and area/power efficiency requires selecting appropriate levels of parallelism and operating frequency for each tile, while also considering that uniformly increasing performance across all components can introduce interconnect bottlenecks.

In practice, these optimization steps are rarely fully automated. Most SoC designs still rely on manual decision-making, supported by heuristics or partial automation. A common approach is DSE, in which a subset of possible configurations is systematically evaluated to identify Pareto-optimal trade-offs between performance, power, and area. While DSE tools have matured, the sheer size of the design space, coupled with heterogeneous workloads, makes exhaustive exploration impractical, leaving expert intuition and heuristic-driven pruning essential to the process.

A generic flow for the DSE of tile-based, heterogeneous, multi-core SoCs meant for deployment on FPGA targets is depicted in Figure 2.2. Each iteration of the DSE process selects first a new configuration to evaluate, identifying the computing tile to place in each available position of the target computing platform. The RTL description of the SoC and of the

selected tiles assigned to the various nodes in the NoC interconnect are then synthesized and implemented to generate a bitstream. The latter is flashed to an FPGA chip, and running the execution of a workload defined for its performance evaluation allows collecting performance-related statistics of the current SoC configuration. Such process is repeated for each design configuration explored in the DSE.

The synthesis and implementation step is notably by far the most time-consuming, and repeating it for each single point to be visited in the design space makes it impossible from a practical standpoint to perform a complete DSE of even small-sized heterogeneous multi-core SoC computing platforms to identify their globally optimal configurations.

2.3 Dynamic partial reconfiguration

DPR enables portions of the FPGA fabric to be reconfigured at run time without interrupting the operation of the remaining regions of the device.

DPR operates by dividing the FPGA design into static regions, which remain unchanged, and reconfigurable regions, which can be dynamically modified. The static region typically contains essential system components, e.g., control logic, memory interfaces, and communication protocols, that must operate continuously. The reconfigurable regions are independent portions of the FPGA that can be swapped with alternative modules, allowing changes to the hardware's functionality. This is achieved by loading precompiled configuration data, referred to as partial bitstreams, into the reconfigurable regions. These bitstreams are managed and transferred via dedicated configuration interfaces, ensuring uninterrupted operation of the static logic during reconfiguration. Reconfigurable regions must notably be constrained during the floorplanning phase to specific areas of the device, thus avoiding any overlaps with static logic and any resource conflicts.

The DPR design flow begins with partitioning the FPGA design into static and reconfigurable regions, allocating specific areas of the FPGA fabric for each region. The static region contains logic that must remain operational during reconfiguration, while the reconfigurable regions are allocated specific resources for dynamic updates. Each reconfigurable module corresponds to a well-defined portion of the RTL hierarchy. These modules are designed independently, with consistent interfaces to ensure compatibility between different configurations. This consistency is essential for enabling the substitution of one module with another, as all modules that can occupy the same reconfigurable region must adhere to the same interface specifications.

Once the modules are defined, the next step is to generate configuration files, producing a full bitstream that contains the configuration for the entire design and partial bitstreams for each alternative configuration of the reconfigurable regions. These partial bitstreams allow for run-time updates to the reconfigurable regions without affecting the static logic.

The substitution of RTL modules is managed at run time by a control mechanism from either the host PC or a processor in the static region, which selects the appropriate partial bitstream and initiates the reconfiguration process by transferring the selected bitstream into the FPGA's configuration memory without halting the static logic, which enables continuous operation of the system during reconfiguration.

CHAPTER 3

Related works

This chapter reviews the body of literature relevant to the topics addressed in this research, which provides the foundation on which this thesis is built and against which its contributions are evaluated. To this end, the chapter first discusses the most widely adopted academic tools for SoC design and prototyping, with particular attention to ESP, whose architecture serves as the foundation for the proposed SoC prototyping framework. The discussion then turns to the literature on heterogeneous SoC optimization, covering key aspects such as accelerator allocation, hardware/software co-design, and DSE methodologies. Finally, the chapter also surveys some FPGA-oriented features that have been employed in the proposed SoC prototyping platform, namely DFS architectures, DPR frameworks, and memory benchmarking platforms.

The structure of the chapter is as follows. Section 3.1 introduces the principal SoC prototyping platforms found in the literature, with a detailed discussion of the ESP architecture. Section 3.2 examines the most relevant optimization methodologies applied to heterogeneous SoCs. Section 3.3 discusses several DFS architectures applied to SoCs. Section 3.4 presents recent examples of DPR applications drawn from academic studies. Finally,

Table 3.1: *Comparison between Omega and the main SoC platforms from the state of the art.*

	PULP [17]	Chipyard [13]	ESP [68]	**Omega**
Full prototyping flow	✗	✓	✓	✓
DFS	✗	✗	✗	✓
Multi-replica tiles	✗	✗	✗	✓
Memory benchmarking	✗	✗	✗	✓
DPR	✗	✗	✓[1]	✓

3.5 analyzes the two most important memory benchmarking platforms for DDR and HBM memories targeting FPGA devices.

3.1 SoC prototyping frameworks

Several frameworks have been developed for the design of heterogeneous multi-core SoCs, taking advantage in recent years of the growing adoption of the RISC-V ISA and the flexibility of FPGA platforms. FPGAs, in particular, have proven well suited to prototype-based design methodologies due to their reconfigurability and ability to support rapid hardware/software (HW/SW) co-design.

The HERO [61] research framework combines an ARM CPU with many soft-core RISC-V processors instantiated on an FPGA, enabling fast prototyping and verification of accelerators using OpenMP. The Celerity project [32] enabled reducing development time of a modular, massive SoC by structuring it into three tiers, namely, a general-purpose one with Linux-capable RISC-V cores, a parallel and scalable RISC-V array, and an HLS-based neural network accelerator, to facilitate its design. [55] proposed, with the goal of facilitating the hardware design flow, an object-oriented HLS tool with its own HLS library and based on the use of latency-insensitive channels and a globally asynchronous locally synchronous (GALS) clock architecture to improve the timing of large circuits. The BlackParrot [85] agile RISC-V multi-core heterogeneous SoC design framework targets the modular design of a computing platform capable of running the Linux operating system and integrating CPU cores and accelerators while maintaining cache coherence. Other recent tile-based heterogeneous multi- and many-core platforms pairing RISC-V general-purpose CPU cores with dedicated hardware accelerators include the AGILER [51] and PEAK [102] SoCs, which also are meant for deployment on FPGA targets.

[1] The DPR infrastructure is not available in the open-source repository of ESP, but it has been described in [92].

Among the various frameworks proposed in this area, three have been especially influential in shaping the field: ESP [68], Chipyard [13], and PULP [17]. Each of these frameworks is therefore discussed in its own dedicated subsection. Table 3.1 summarizes the different features available in these SoC platforms, providing also a comparison with the Omega framework presented in this thesis.

3.1.1 ESP

The ESP framework for complex heterogeneous SoC design and programming combines a scalable architecture and a flexible system-level design methodology to deliver prototypes targeting FPGAs [68]. Its SoC architecture consists of a set of computing tiles arranged in a 2D mesh and connected through a multi-plane NoC with X-Y routing, as depicted in Figure 3.1. A complementary software framework facilitates the quick and flexible generation of several variants of the computing platform.

The SoC architecture decouples the design of the computing tiles and of the NoC interconnect, according to the principles of communication-based system-level design. The NoC interconnect is synchronous and achieves single-cycle latency between adjacent routers. Each tile implements a bridge to convert bus requests to NoC messages and vice versa, enabling the components in different tiles to interact as if they were connected to the same bus.

Each tile implements either a CPU core (*CPU* in Figure 3.1), a memory controller (*MEM*), a hardware accelerator (*ACC*), or an auxiliary module (*I/O*).

The *CPU* tiles can instantiate the SPARC V8 LEON3 [101] core or the RISC-V CVA6 [112] and Ibex [89] ones, as well as L2 cache to transparently participate in the ESP coherence protocol thus supporting multi-core execution and Linux's SMP kernel boot. The *MEM* tiles instantiate controllers for DDR3 or DDR4 memory. The *ACC* tiles host accelerators obtained through ESP-supported design flows from their RTL, C, or SystemC descriptions, as well as any third-party cores that expose a standard AXI interface [39]. The ESP framework handles address translation, DMA, configuration registers, and coherence to enable hardware designers focusing on the accelerators' optimization rather than on their integration into the SoC. The *I/O* tile hosts the non-memory I/O interfaces of the SoC, e.g., Ethernet and UART, and other components such as the boot ROM and interrupt controller.

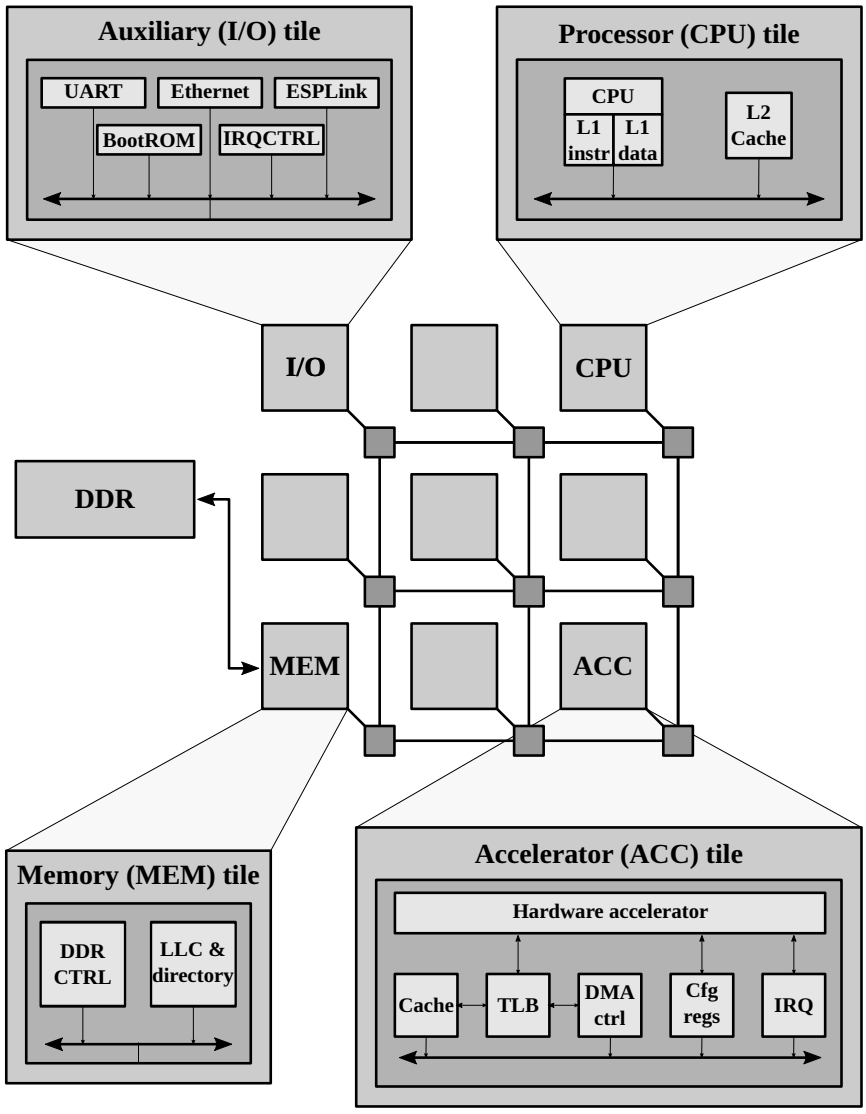


Figure 3.1: *Schematic view of a SoC generated with open ESP.*

3.1.2 Chipyard

Chipyard [13] is an integrated environment for SoC design, simulation, and implementation. It leverages the Chisel hardware construction language [19] to describe highly modular SoCs and to customize the interconnections among their many components. These components can either be developed from scratch or selected from a library of IP blocks, which can be further customized through parameterized IP generators. To this end, Chipyard provides a collection of hardware modules that can be instantiated in generated SoCs, ranging from general-purpose RISC-V processors, such as Rocket [16] (a five-stage, in-order scalar CPU) and BOOM [116] (a 64-bit, seven-stage out-of-order CPU), to domain-specific accelerators like Hwacha and Gemmini [38].

The modularity of Chipyard is enabled by Chisel [19], a hardware construction language embedded within Scala. Chisel introduces higher-level programming abstractions compared to traditional hardware description languages (HDLs) such as Verilog and VHDL. It supports object-oriented and functional paradigms, as well as type parametrization and inference–features that make it particularly well-suited for developing IP generators capable of producing configurable IP instances. However, the use of Chisel also presents a limitation to the broader adoption of Chipyard. The language's specialized nature means that expertise in Chisel is less widespread than for conventional HDLs, and support among existing simulators remains limited.

Once the SoC design described in Chisel is translated into an RTL representation, Chipyard can generate three distinct downstream outputs, each tailored for a different hardware development flow: traditional RTL simulation, FPGA-accelerated simulation with FireSim [54], and physical design using the Hammer VLSI flow [64]. The RTL simulation flow supports both commercial (Synopsys VCS) and open-source (Veripool Verilator) simulators. FireSim [54] is an open-source framework for full-system FPGA-based simulation. It supports both local FPGA platforms and AWS EC2 F1 instances, achieving performance improvements of more than an order of magnitude over software-only simulation. Hammer [64] is an open-source, reusable physical design flow generator that reduces design effort and improves portability. It employs a modular architecture that decouples the major aspects of the VLSI flow (design, tool, and process technology), enabling a flexible and reusable implementation framework.

3.1.3 PULP

The PULP (Parallel Ultra-Low Power) project [17] aims to develop open-source, RISC-V-based multi-core SoCs that deliver increased computational capability for IoT and embedded applications while maintaining power consumption in the milliwatt range. This efficiency is achieved through extensive parallelization, employing multiple lightweight RISC-V cores in combination with domain-specific accelerators. Rather than a single unified SoC framework, the PULP ecosystem consists of a collection of components and SoC designs, each tailored for specific application domains.

CPUs play a central role within the PULP environment. The platform currently supports five RISC-V processors—CV32E40P, Ibex, Micro-riscy, CVA6, and Snitch [37, 89, 112, 114]—each optimized for a distinct use case. At one end of the spectrum, the 32-bit, single-stage Snitch core [114] prioritizes maximum energy efficiency and minimal silicon area, relying on an external floating-point unit. At the other end, the 64-bit, application-class CVA6 processor [112] supports multiple RISC-V extensions and is capable of running a full Linux operating system.

Several SoC designs have been developed within the PULP framework [30, 80, 83, 90, 113]. All of them integrate one or more PULP cores interconnected through specialized AXI-based crossbars and NoCs. Their target applications range from IoT end-node systems [30, 90] and high-efficiency floating-point computation [113], to heterogeneous designs combining general-purpose CPUs with domain-specific accelerators [80], and machine-learning workloads employing vector-processing units [83].

However, despite leveraging a PULP IP block (the CVA6 core), the SoC generator developed in this thesis does not adopt the PULP platform as its foundation. This is primarily due to two reasons. First, PULP's ultra-low-power orientation conflicts with the performance and power targets of data center-class platforms considered in this work. Second, the PULP ecosystem is not a unified SoC prototyping framework but rather a collection of compatible IPs and pre-designed SoCs. As such, it cannot directly support the kind of DSE optimization processes that are the focus of this thesis.

3.2 SoC optimization

DSE mechanisms have been the focus of vast research to face the increasing system complexity of multi-core SoCs and help the designers in the optimization of their architectures. Early works make use of genetic algorithms to deliver multi-objective optimization targeting SoCs with general-purpose

Table 3.2: *Comparison between FARMER and the main SoC optimization methodologies from the state of the art. Legend:* ***MILP*** *mixed-integer linear programming,* ***GA*** *genetic algorithm,* ***TPE*** *tree parzen estimators,* ***GP*** *gaussian process.*

	[31]	[25]	[111]	[29]	[49]	**FARMER**
Algorithm	MILP	Custom	GA	TPE	GA	GP
Full system	✓	✓	✓	✓	✗	✓
Iterative	✗	✗	✓	✓	✗	✓
On-chip testing	✗	✗	✓	✗	✗	✓
Learning-based	✗	✗	✗	✗	✗	✓

CPU cores [46, 52, 81]. However, these proposals focus on homogeneous platforms and thus they are unsuitable for heterogeneous SoCs with domain-specific accelerators as the ones considered in this thesis.

3.2.1 HW/SW co-design

The integration of hardware accelerators both in embedded systems and in HPC has made it critical to efficiently explore the ever-larger design space of complex, heterogeneous SoCs. The ensuing research efforts led to the development of novel approaches for hardware/software co-design [26, 86, 88, 117].

Hardware/software co-design frameworks partition the target applications into tasks to be accelerated in hardware and tasks to be executed in software by general-purpose CPU cores [117]. The emergence of HLS frameworks, that allow for a fast generation of an RTL description of kernels originally written in high-level languages such as C/C++ and SystemC, has enabled the automation of the design and implementation of hardware accelerators starting from their corresponding software code, leaving the researchers with the task of identifying the code regions that benefit the most from acceleration.

[117] proposed an approach based on the identification of so-called hot loops of an application, i.e., the loops that take up more than 1% of the total running time, to reduce the exploration space on which to run their DSE. Other works focus on other aspects of the applications under execution, ranging from the memory dependencies between code regions [86] to the interaction of HW/SW partitioning with the optimization of HLS pragmas [88] and the reuse of the same accelerators for different parts of the applications [26].

3.2.2 Accelerator mapping

Accelerator mapping frameworks target instead the design of heterogeneous SoCs whose hardware accelerators are already available to the designers. Such frameworks are tasked therefore with identifying the optimal allocation of the accelerators by choosing the number of their instances, their position, and their assignment to specific CPU cores and memory channels. The incapability of exhaustively exploring the large design space has led to the development of models and heuristic-based approaches to accelerate the DSE by reducing the number of SoC configurations to be actually explored. [31] employs a mixed-integer linear programming (MILP) approach to explore the large design space, while [25] focuses on identifying an ad-hoc metric for the optimal allocation of shared accelerators to the CPU cores in a heterogeneous SoC. Other works focus on automating the DSE to reduce its time-to-solution. [111] proposes a two-step automated DSE flow split into configuration generation and SoC evaluation, the latter of whom can be either a standard cycle-accurate simulation, an accelerated simulation where the accelerator is replaced with a model, or a further accelerated evaluation that tests the SoC on an FPGA board. Finally, research has been devoted to improving the quality of models of SoCs to be explored by leveraging the actual results of the evaluated SoC instances. [29] introduces an iterative approach to optimize a heterogeneous SoC, using a hyperparameter-based model that is improved with the statistics collected during the simulation of each configuration, thus driving a better choice of the next SoC instance to test.

A line of research proposes HW or HW/SW schedulers [104] [103] [110]. Such an approach, while being more versatile for rapidly changing applications, is not suitable for the consolidation of more complex workloads that remain stable over a long period of time. In [104], Vaishnav et al. define the concept of *resource elastic virtualization*, which represents the capability of a system to resize and reallocate hardware tasks as needed. The authors developed a HW scheduler, a tool that can allocate tasks at run-time on an FPGA to minimize the execution time of each task. They further extend this methodology by adding the possibility to schedule and execute tasks also on a multi-core CPU, making it viable for heterogeneous SoCs [103]. A HW/SW scheduler is also proposed in [110], where the authors highlight the importance of assessing the communication profile of an application (that is, if it is memory- or compute-bound) to maximize the efficiency of the algorithm.

Table 3.3: *Comparison between the proposed DFS infrastructure and the main DFS and DVFS implementations from the state of the art. Legend:* ***PLL*** *phase-locked loop,* ***DCM*** *dynamic clock management,* ***BC*** *buck converter,* ***IVR*** *integrated voltage regulator,* ***MMCM*** *mixed-mode clock manager,* ***PMIC*** *power management integrated circuit.*

	[87]	[41]	[59]	[42]	[67]	[47]	**Our**
Technology	PLL	PLL	DCM	BC	IVR + PLL	MMCM + PMIC	MMCM
Voltage scaling	✓	✓	✗	✓	✓	✓	✗
Frequency scaling	✓	✓	✓	✓	✓	✓	✓
FPGA-based	✗	✗	✓	✗	✓	✓	✓
NoC-based	✗	✓	✓	✓	✓	✗	✓
On-chip testing	✗	✗	✓	✗	✓	✓	✓
Glitch-free	✓	✓	✓	✓	✓	✗	✓
Fine granularity	✗	✗	✗	✗	✗	✓	✓

Other works instead focus on specific types of applications, like [94]. Therein, the authors define an area-performance optimization problem for multi-kernel applications (e.g., convolutional neural networks), where each kernel can be instantiated with a certain level of parallelism and on different FPGAs in a cluster. Such a problem is solved using a heuristic composed of two steps: identifying the number of computational units (CUs) for each kernel (without exceeding the resource constraints) and placing such CUs on the FPGAs, minimizing the spread of CUs belonging to the same kernel.

The problem of allocating an entire set of applications was first addressed by Jordan et al. [49]. In this work, the authors propose an algorithm based on the *genetic multidimensional knapsack* solution to rapidly identify the best task deployment on a heterogeneous system. The algorithm is also tunable to select the optimal area-performance trade-off. This solution, however, relies on a complete knowledge of the incoming kernels and does not take into account different levels of parallelization.

Table 3.2 compares the main acceleration mapping frameworks available in the state of the art with the proposed FARMER optimization methodology, in terms of employed algorithm and functionalities.

3.3 Dynamic voltage and frequency scaling

The literature provides a variety of run-time optimization policies leveraging dynamic voltage and frequency scaling (DVFS) actuators in complex heterogeneous SoCs [24, 41, 42, 47, 59, 67, 87], whose functionalities and characteristics are summarized in Table 3.3. For example, the work in [41]

shows how DVFS can effectively exploit the different energetic profiles due to process variability of processors implemented on the same die, obtaining a reduction of the power consumption of the chip, while DFS is shown to be particularly effective on NoC-based systems [24, 42, 59]. The proposal in [67] remarkedly highlights the role of DVFS when exploring the design space of complex SoCs, being able to reduce the energy consumption of the hardware accelerators by up to 85%. However, none of the SoC prototyping frameworks from the state of the art listed in Section 3.1 includes the design of a flexible DFS architecture to explore run-time optimizations of the SoCs.

3.4 Dynamic partial reconfiguration

DPR has been available in commercial FPGAs since the 2000s [105], and has been exploited both in embedded systems, where it allows time-multiplexing several functionalities in the same resource-constrained chip [93], and by FPGA-as-a-service cloud providers, enabling the allocation of new users' applications on the same multi-tenant device without interfering with the other users' workloads [12].

While scarcely adopted by the industry due to the many low-level details to consider in reconfigurable designs [105], the open literature includes tools that automate the design of DPR applications, e.g., generating the floorplan and carrying out the implementation [93]. Other works target instead specific use cases, such as reducing the number of partial reconfigurations on multi-tenant FPGAs [57].

The DPR functionality was also integrated in few SoC frameworks like PR-ESP [92], that focuses on reducing the implementation time of reconfigurable designs through a heuristic that determines the optimal level of parallelization for the implementation process depending on the relative size of the static and dynamic parts of the design.

Other solutions from the open literature leverage the capability of dividing an FPGA into separate reconfigurable regions in order to split large designs into modular blocks that can be processed independently and consequently reduce the implementation times compared to the baseline monolithic designs. RapidWright [63] modifies the flow of the AMD Vivado EDA toolchain to reuse RTL modules implemented in isolation, improving both the timing of the overall design and the total implementation time, while [109] divides the HLS design into functional blocks that are mapped to different reconfigurable regions and reimplemented only when needed, similarly to how in the traditional software domain only the code that has been actually modified is recompiled. Other works [108, 109] introduced the

Table 3.4: *Comparison between the proposed memory benchmarking platform and similar works from the state of the art.*

	[66]	[43]	[79]	[82]	**Our**
Full-system evaluation	✓	✓	✗	✓	✓
Performance collection	✓	✓	✗	✓	✓
Finely-tunable traffic	✗	✓	✗	✓	✓
Mixed R/W traffic	✗	✗	✗	✗	✓
Data coherence check	✗	✗	✓	✓	✓

concept, typical of software flows, of re-compiling only the code that have been actually modified, by dividing the HLS design into functional blocks that are mapped to different reconfigurable regions and re-implemented only when needed.

3.5 Memory benchmarking platforms

The open literature provides a number of solutions that target the evaluation of memory systems [43, 66, 79, 82], which are compared in Table 3.4. Shuhai [43] delivers a benchmarking platform for the performance of HBM2 and DDR4 memories on FPGAs. However, it avoids checking data coherence by constantly providing zeros as data to be written, its workloads are limited to read-only and write-only operations with sequential, i.e., non-random, addressing, and it employs a fixed access pattern typically used in FPGA workloads with configurable stride and working size. Moreover, Shuhai is only compatible with boards that feature a PCIe connection to the host, particularly AMD Alveo cards, and depends on an OS driver. Conversely, DRAM Bender [79] provides an infrastructure for standalone testing and characterization of HBM2 and DDR4 memories, while disregarding the broader system context. It leverages a memory controller with a custom instruction set and general-purpose registers to give its user maximum programmability, enabling fine-tuned control to support thorough analyses related to physical security aspects, e.g., Rowhammer attacks [58, 77], which are the main target of DRAM Bender.

CHAPTER 4

Omega SoC prototyping platform

This chapter introduces Omega, a DSE-oriented SoC prototyping platform. Omega enables rapid exploration of large configuration spaces through three main advancements over existing SoC prototyping frameworks. The first is flexibility, which provides extensive configuration options and fine-grained control over PPA trade-offs. The second is reconfigurability, allowing runtime modification of multiple system characteristics. This capability supports the evaluation of diverse SoC configurations without re-initiating the implementation process, thereby accelerating the exploration workflow. The third is observability, which offers comprehensive monitoring and benchmarking features to assess system behavior and effectively guide the design space exploration process.

The rest of this chapter, after an overview of the Omega framework in Section 4.1, describes the components that differentiate Omega from other SoC prototyping tools, namely the DFS mechanism in Section 4.2, the multi-replica accelerator-tile architecture in Section 4.3, the DPR functionality to dynamically swap accelerators in Section 4.4, the memory benchmarking interface in Section 4.5, and the monitoring infrastructure in Section 4.6. Finally, Section 4.7 depicts the full DSE flow of the Omega framework.

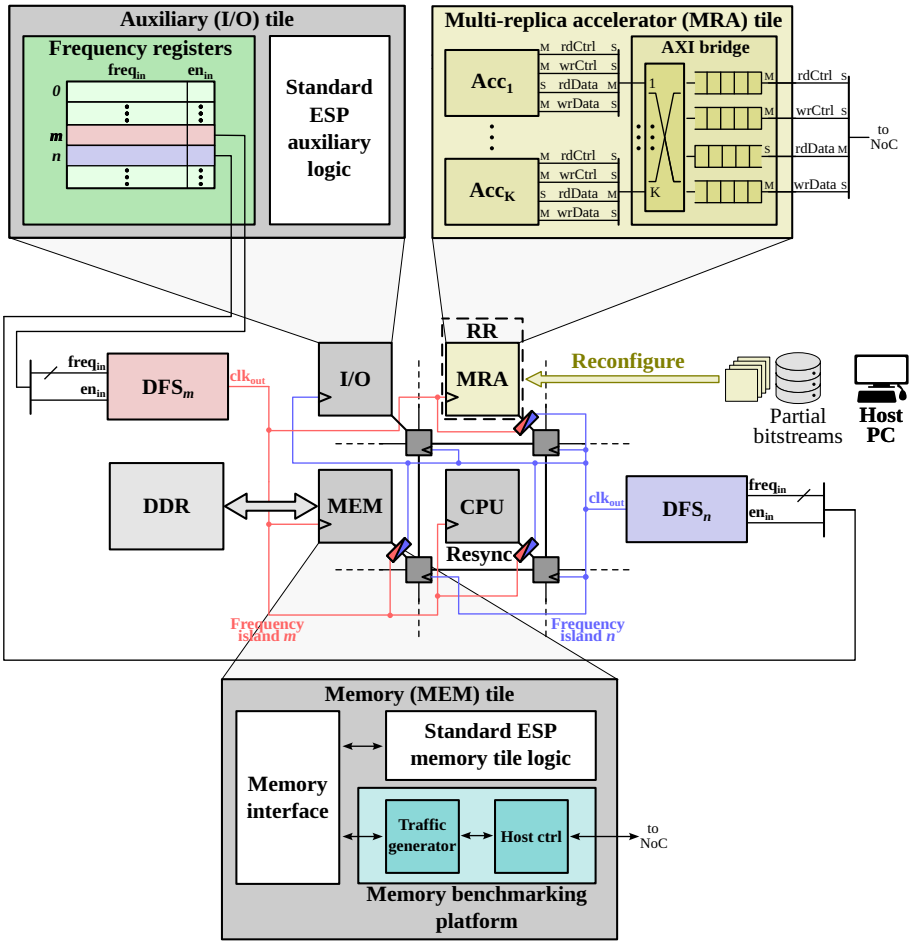


Figure 4.1: *Architecture of a generic SoC obtained through the Omega framework.*

4.1 Architecture overview

The Omega framework provides the system designers with the ability to easily generate, through a highly configurable template architecture, accelerator-based heterogeneous multi-core SoC designs that target FPGA deployment. The SoC architecture, depicted in Figure 4.1, features the capabilities of *(i)* dynamically swapping the accelerator tiles at run time by leveraging the DPR technology of the target FPGAs, *(ii)* instantiating multiple identical accelerators in the same accelerator tile to scale their throughput, and *(iii)* collecting performance-related statistics via a dedicated run-time monitoring infrastructure. Omega's SoC architecture consists of tiles arranged in a 2D mesh and connected through a synchronous NoC interconnect with single-cycle latency routers, 64-bit data bandwidth, and deterministic X-Y routing.

The SoC includes a number of reconfigurable regions (RR in Figure 4.1) that can feature any of the accelerator tiles or be left empty, and the accelerator and empty tiles can be dynamically swapped with new ones by flashing the corresponding partial bitstreams from the host PC.

Each accelerator tile (ACC in Figure 4.1) hosts a set of K replicas of an accelerator's computational cores, where K is a configurable parameter defined by the system designer and that enables easily tuning the trade-off between throughput and area. The ACC tiles can instantiate accelerators obtained through HLS, hardware accelerators described in RTL, and generic IP cores that expose an AXI4-Stream interface. The CPU tiles include instead 64-bit RISC-V CVA6 [112] CPU cores, the MEM tiles instantiate memory controllers, and the I/O tile hosts the non-memory I/O interfaces of the SoC, e.g., Ethernet and UART, and other components such as the boot ROM and interrupt controller.

The run-time monitoring infrastructure optionally implements, for each tile, specific counters devoted to measuring the execution time, the round-trip time of data packets, and the number of incoming and outgoing packets. Tiles enhanced with such run-time monitoring capabilities expose their statistics via memory-mapped registers that can be accessed both via software executing on CPU cores of the SoC itself and via the USB-to-serial interface that connects the platform to the host.

A complementary software framework facilitates the quick and flexible generation of Omega SoCs with different number of tiles, different NoC topologies, and any possible combination of the various computing, memory, and I/O tiles.

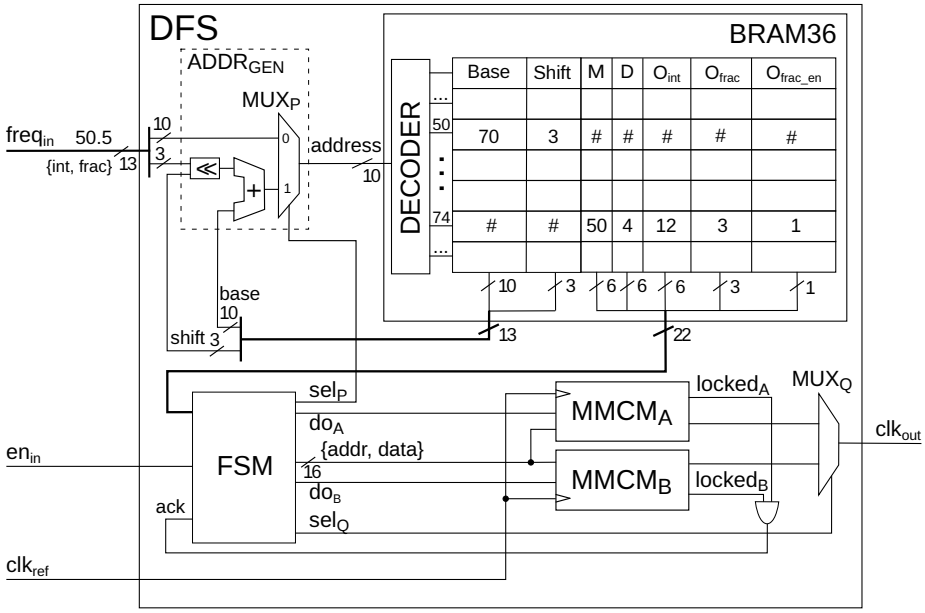


Figure 4.2: *Architecture of the proposed DFS actuator.*

4.2 Configurable-DFS frequency-island architecture

The proposed framework allows partitioning the SoC into multiple frequency islands at design time. Every SoC tile and NoC router is assigned to a frequency island that can group multiple computing and routing elements. Each frequency island is fed an independent clock signal that can either have a fixed frequency or be generated by a dedicated DFS actuator.

The architecture supporting such SoC partitioning can be split into three main components, namely a set of registers holding the frequency configuration of each island (*Frequency registers* in Figure 4.1), a DFS actuator (*DFS*) for each of the frequency islands, and resynchronizers (*Resync*) at the boundaries of the latter.

The dedicated *Frequency registers* memory is added to the I/O tile to store the frequency configuration of each island in the SoC. A register for each frequency island holds two fields encoding, respectively, the current clock frequency and an enable flag signaling whether the DFS actuator should reconfigure its output clock signal to match such frequency value. The registers are memory-mapped, simplifying the access when actuating a frequency reconfiguration or retrieving the current clock frequency of a specific island, e.g., in the context of a power or energy management scheme. For example, the reconfiguration of frequency island FI_m to clock frequency

f^* can be triggered by performing two subsequent write operations, one writing f^* at the memory address of FI_m's clock frequency ($freq_{in}$) register field and another writing 1 at the address of FI_m's reconfiguration enable (en_{in}) one. A dedicated line between each island frequency register and its corresponding DFS delivers the clock frequencies and enable flags to the frequency islands for their actuation. Once the clock reconfiguration has been triggered, the corresponding enable flag in the register is reset back to 0, until an external actor in charge of the clock frequency management enforces another reconfiguration.

The resynchronization infrastructure equips the tile-based processor with resynchronizers to ensure the correct functionality between components, i.e., both NoC routers and SoC tiles, located on different frequency islands. In particular, since our framework targets AMD FPGAs, each resynchronizer is implemented as a dual-clock first-in-first-out (FIFO) buffer, fed with the clock signals of the two islands it connects.

The mixed-mode clock managers (MMCMs) included in AMD FPGAs can generate clock signals with a defined frequency from an input reference clock. The parameters determining the output frequency are notably modifiable at run time via the dynamic reconfiguration port (DRP). The relationship between the input (f_{in}) and output (f_{out}) clock frequencies of an MMCM primitive is $f_{out} = \frac{M}{D \cdot O} \cdot f_{in}$, where M is the clock feedback multiplier while D and O are clock dividers. Notably, O is usually split into integer O_{int} and fractional O_{frac} parts. The proposed DFS actuator leverages the MMCM to be dynamically reconfigured through the DRP, thus producing an output clock signal with a frequency that can be defined at run time. The ***DFS*** module, whose architecture is depicted in Figure 4.2, receives as its inputs a target clock frequency $freq_{in}$, a reference clock signal clk_{ref}, and a control signal en_{in} to enable the actuation of the target clock frequency $freq_{in}$, and it outputs the generated clock signal clk_{out}. The architecture of the ***DFS*** module can be split into three main parts. The configuration memory ($BRAM36$) contains the M, D, and O parameters for the generation of clk_{out} from clk_{ref}, two MMCM modules $MMCM_A$ and $MMCM_B$ are tasked with the actual generation of clk_{out}, and the finite-state machine (FSM) and related address generation ($ADDR_{GEN}$) logic implement the procedure to perform the dynamic reconfiguration of the clock frequency.

Notably, the clock signal output by the MMCMs of an AMD FPGA remains low during its reconfiguration, thus causing a clock-gating effect on the computing platform. The proposed DFS actuator avoids such negative effect by employing two MMCM components. The FSM forces the master

Algorithm 1 DFS finite state machine algorithm.

1: $\{Master, Slave\} \leftarrow \{MMCM_A, MMCM_B\}$
2: **procedure** RECONFIGURE ($freq_{in}$, en_{in}, ack)
3: **if** $en_{in} \wedge ack$ **then**
4: $config_f \leftarrow$ RETRIEVECONFIG ($freq_{in}$)
5: LOADCONFIG ($config_f$)
6: DOCONFIG ($Slave$)
7: **while** $ack \neq 1$ **do**
8: WAIT ()
9: **end while**
10: $\{Master, Slave\} \leftarrow \{Slave, Master\}$
11: **end if**
12: **end procedure**

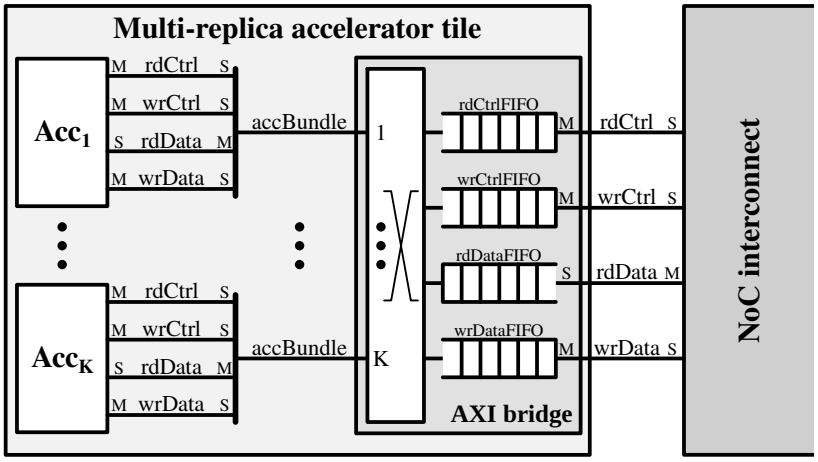


Figure 4.3: *Detailed scheme of the multi-replica accelerator-tile architecture.*

MMCM to preserve the current output clock signal while the slave one is under reconfiguration, after which their roles are swapped.

4.3 Multi-replica accelerator-tile architecture

A simple and time-efficient solution to increase the throughput of an existing hardware accelerator integrated into a computing platform is to instantiate multiple replicas of the former. Indeed, instantiating *K* such replicas intuitively provides a *K*× increase of the throughput of the baseline accelerator.

The proposed multi-replica accelerator tiles (*MRA* in Figure 4.1) can instantiate multiple replicas (Acc_i) of a hardware accelerator within a single computing tile, without any changes either to the accelerator or the size

and topology of the NoC interconnect, and avoiding being forced to occupy multiple computing tiles with the same accelerator in order to improve its performance.

The multi-replica accelerator tiles support any third-party hardware accelerator that exposes an AXI interface. Their replication factor *K*, which corresponds to the number of copies of an accelerator that are instantiated and operate in parallel, is configurable by the system designer to easily tune the trade-off between throughput and area.

A baseline accelerator to be integrated into an Omega SoC exposes four AXI4-Stream interfaces for read control (*rdCtrl*), write control (*wrCtrl*), read data (*rdData*), and write data (*wrData*) purposes, respectively. As shown in Figure 4.3, a multi-replica accelerator tile still exposes the same AXI4-Stream interfaces towards the NoC interconnect (*NoC*) to guarantee full compatibility with the Omega SoC architecture. The *AXI bridge* component is therefore tasked with multiplexing the four AXI4-Stream interfaces of each of the *K* accelerator replicas into four corresponding buffers for the AXI4-Stream interfaces of the tile. It also includes arbitration logic for dispatching contention between concurrent transactions from and to the *K* accelerators to maximize their overall throughput while guaranteeing fairness to each of them. FIFO buffers enable aggregating the transactions from and to the *K* replicas into bursts in order to efficiently use the NoC resources and minimize the impact on the overall traffic.

4.4 DPR mechanism

The capability of swapping the accelerator tiles at run time is enabled by few dedicated hardware-software mechanisms, which are depicted in Figure 4.4. The accelerator tiles all provide the same interface with the NoC interconnect, to effectively enable swapping them with each other in each of the reconfigurable regions. The behavior of clock, reset, and data bus signals at the boundaries between the accelerator tiles in the reconfigurable regions and the NoC interconnect must also be carefully managed, since any signal oscillation at the I/O of a reconfigurable region during the flashing of the partial bitstream would otherwise produce unpredictable results and possibly cause the accelerator tile to malfunction.

Each reconfigurable region of the Omega SoC is therefore connected with the NoC interconnect through its own dedicated DPR interface buffer, which is instantiated in the static region of the FPGA and that avoids oscillations and glitches during DPR through its dual buffering action, by clock-gating the reconfigurable region, asserting the reset signal, and setting all the

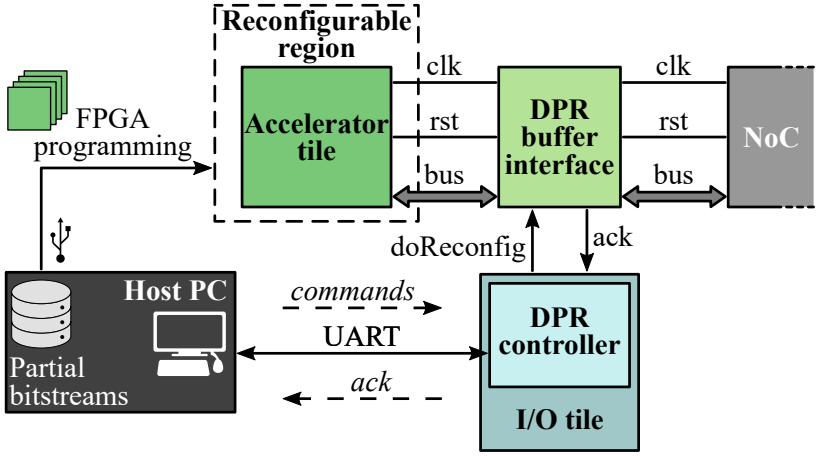


Figure 4.4: *Detailed overview of Omega's architecture for dynamically swapping accelerator tiles.*

incoming signals of the data bus in a reset state. The input clock, reset, and data bus signals are buffered depending on an enable signal (doReconfig in Figure 4.4) that is sent from a dedicated DPR controller module, which is instantiated in the I/O tile of the Omega SoC and manages the reconfiguration of all the reconfigurable regions in the SoC.

The DPR controller module in the I/O tile receives commands from the host PC to drive the DPR of the reconfigurable regions in the SoC and acts accordingly to control the buffering logic at the boundaries between the reconfigurable regions and the NoC interconnect. The host PC holds the partial bitstreams for the tiles to be instantiated in the various reconfigurable regions, and it is connected via UART to the DPR controller to manage the reconfiguration and via USB to the FPGA chip to flash the bitstreams and program the FPGA itself.

Reconfiguration is performed as a sequence of five steps:

1. the host PC initiates a reconfiguration request by indicating the tiles to reconfigure to the DPR controller component via a dedicated command;
2. the DPR controller sends then a reconfiguration signal to the DPR interface buffers connected to the accelerator tiles that have to be swapped with new ones, in order to gate the clock signal fed to the tiles and hold the reset and data bus signals at steady values;
3. the DPR controller sends an acknowledgment response to the host PC, to signal that the latter can initiate the actual reconfiguration of the FPGA chip;

4. the host PC ships the partial bitstreams corresponding to the accelerator tiles to swap in through its USB connection to the FPGA chip, and such bitstreams are accordingly flashed by the FPGA programming logic;

5. once the DPR process for the tiles to be updated has been completed, the DPR controller module deasserts its reconfiguration flag signal to the enabled DPR interface buffers which stop gating the clock, deassert reset signals, and get back to the normal-behavior buffering of the data bus signals.

4.5 Memory benchmarking

The proposed memory benchmarking platform, meant to be instantiated on an FPGA and whose architecture is depicted in Figure 4.5, can be split into three main components, namely, the memory interfaces, the traffic generators, and a host controller.

Configurability in the generated traffic is supported by the wide variety of memory access patterns implemented by the traffic generator component and is leveraged at run time by driving the desired tests and collecting the corresponding results and statistics through the host controller component. Flexibility in the number of memory channels is achieved by instantiating a memory interface and a traffic generator for each channel, and the memory data rate and performance counters to implement are also defined at design time. Table 4.1 summarizes the parameters that enable configuring the memory benchmarking platform at design time and at run time.

Memory Interface Each memory is connected to an interface component that can be split into a physical layer (PHY) and a memory controller, which operate at clock frequencies that differ according to a specific ratio and can be adjusted in order to support memories with different data rates. The memory controller subcomponent receives as its inputs read and write requests, possibly concurrently, buffers and reorders them to boost performance while maintaining data integrity, and then passes them to the PHY layer, which manages the low-level signaling and data transfer to and from the memory.

By operating at a clock frequency that is higher than the rest of the architecture, the memory interface is able to issue multiple commands to the memory at a time. Considering DDR/HBM memories, this includes activate, column address strobe (CAS), and precharge ones, which prepare a row of memory cells for reading or writing, specify the exact column within the

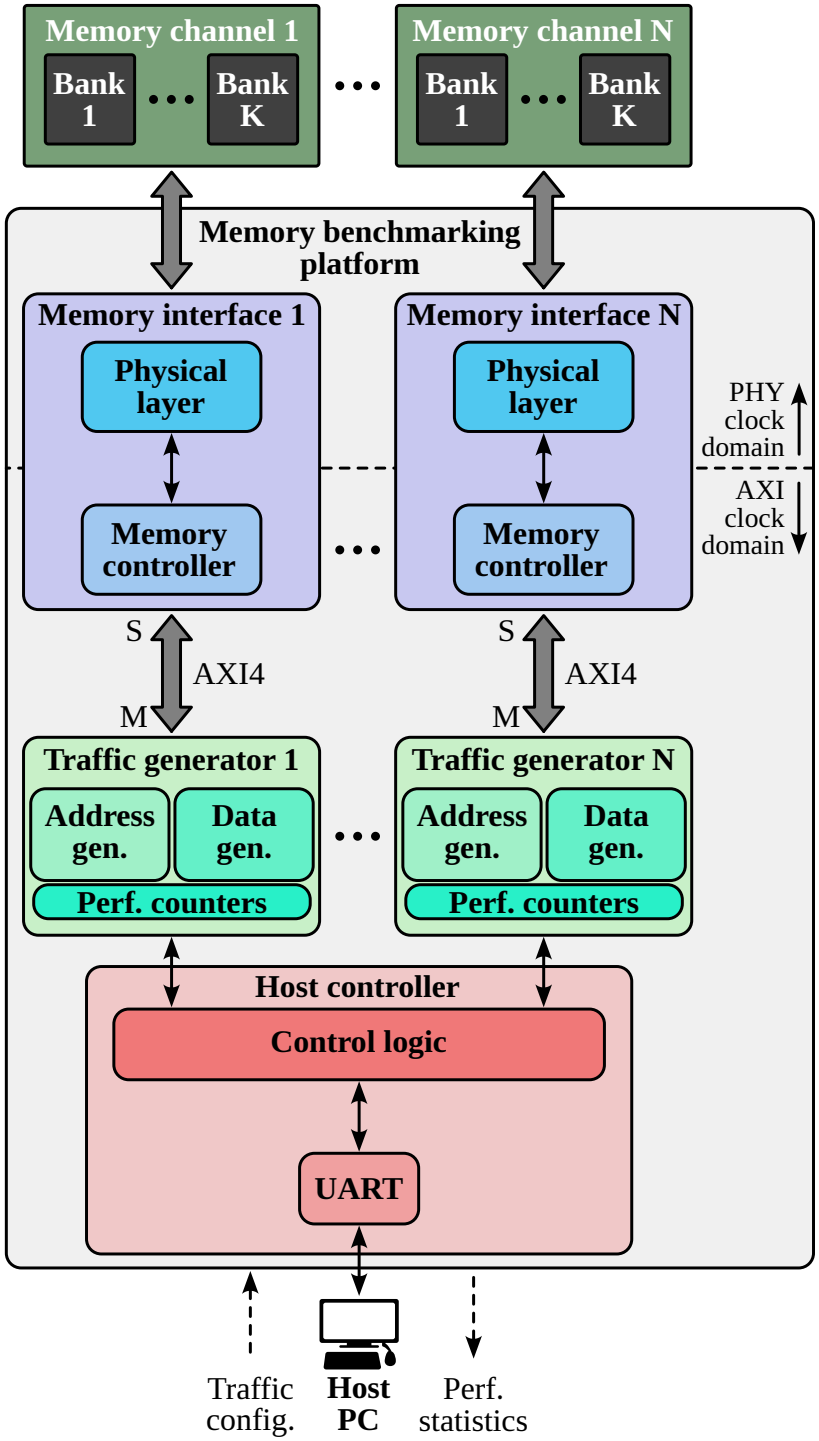


Figure 4.5: *Architecture of the proposed memory benchmarking platform. Legend:* ***S*** *AXI4 slave,* ***M*** *AXI4 master.*

Table 4.1: *Configurability of the memory benchmarking platform.*

Design-time parameters	Run-time parameters
Number of memory channels	Mix of read and write operations
Memory data rate	Sequential or random accesses
Performance counters	Length and type of bursts
	Signaling mode
	Length of transaction batches

activated row to read from or write to, and close the current row of memory making it ready for a new access operation, respectively. Reading from and writing to memory cells involves indeed several steps to leverage the parallel operations and improved efficiency enabled by the hierarchical structure of the memory [44].

The memory controller is also tasked with managing the periodic refresh operations that allow maintaining data integrity in volatile memories by rewriting the data in each cell and therefore restoring its charge to the correct level.

Traffic Generator The traffic generator (TG) component is tasked with generating read and write transactions according to a wide variety of memory access patterns that can be configured at run time. It manages five independent channels dedicated to the read and write address, read and write data, and write response, according to the specification of the AXI4 on-chip communication bus protocol [15]. The ability to separately and concurrently manage the read- and write-related channels enables simultaneous issuing read and write transactions, maximizing the TG's throughput.

On the address generation side, the TG component supports, for both sequential and random addressing modes, burst transactions with any burst length, i.e., the number of data transfers in the burst, ranging from 1 to 128. Burst transactions allow for multiple data transfers with a single address phase, thus reducing the overhead of address transfers and conversely increasing data transfer efficiency in scenarios where large blocks of data need to be moved. Bursts can be fixed, with a constant address for each data transfer, incrementing, where the address increments by the size of the data for each transfer, and wrapping, similar to the previous one but where the address wraps around on a boundary. Moreover, the TG implements multiple signaling modes, namely, a non-blocking mode that mimics a generic AXI device by issuing new requests as soon as possible, a blocking mode that delays new requests until all outstanding transactions are completed, and an

aggressive mode that emulates a device always asserting the ready signal for immediate data transfers.

On the data generation side, differently from Shuhai [43], TGs generate various sequences of non-zero data and can check the correctness of read data against the previously written one.

Finally, the TG component exposes to the host controller a set of performance monitoring counters, configured at design time and including two counters for the clock cycles taken by batches of read and write memory access transactions, respectively.

Host Controller At run time, each TG component is configured through dedicated commands to generate batches of transactions with solely read and write requests or a mix of them, with single transactions or burst ones with a chosen burst length of the fixed, incrementing, or wrapping type, with random or sequential addressing modes, and with non-blocking, blocking, or aggressive signaling. Such run-time configurable parameters are summarized in the rightmost column of Table 4.1.

The host controller is tasked with configuring independently each instantiated traffic generator according to the user's requests, sent from a host PC through the UART serial connection. In addition, performance-related statistics are sent back to the host PC by making use of the performance counters exposed by the TG components, e.g., the throughput of a batch of transactions on a memory channel is obtained by dividing their overall execution time stored in the corresponding counter by the number of such transactions. Other statistics that can be collected by our platform include latency and refresh-related performance degradation.

4.6 Run-time monitoring infrastructure

The run-time monitoring infrastructure is crucial to collecting statistics from the computing platform and thus supporting run-time optimization policies and design space exploration.

For each accelerator, the run-time monitoring infrastructure can selectively enable the monitoring of up to four different statistics, each corresponding to a specific counter in the tile: execution time, number of ingoing and outgoing packets, and the round-trip time. The latter is defined as the time occurring between a request for data from an accelerator to the main memory through DMA and the ensuing arrival of such data to the accelerator.

The execution time counter automatically resets when the accelerator tile starts its computation and stops when the latter is completed, while the other three counters are reset manually. All counters instantiated on accelerator tiles expose memory-mapped registers that can be accessed both via software executing on CPU cores of the SoC itself and via the USB-to-serial interface that connects the platform to the host.

4.7 Full DSE flow

The Omega framework integrates the features described in the previous sections to support DSE that is both fast and comprehensive. The DFS and multi-replica architecture enable fine-grained control of PPA trade-offs at the tile level, thereby allowing precise adjustment of the overall SoC PPA profile. In addition, the DFS and DPR infrastructures provide runtime reconfigurability, making it possible to evaluate new parameter combinations without regenerating a bitstream from scratch. In particular, DPR facilitates complete reconfiguration of several tiles within the SoC, significantly accelerating the exploration process by shifting the time-consuming synthesis and implementation of the computing platform outside the DSE loop. Finally, the memory benchmarking platform and monitoring interface enable the collection of detailed execution statistics across the entire SoC. These metrics can be used both to pre-select configurations for evaluation or to guide exploration dynamically, for example by integrating a DSE methodology into the feedback loop, such as the one presented in Chapter 5.

The full-system DSE flow of Omega is realized through a hardware-software infrastructure that leverages the hardware architecture of the SoCs and in particular their features related to dynamically swapping the accelerator tiles, as described in Section 4.4, and that interfaces with the EDA toolchain for synthesis and implementation, generation of the SoC bitstream and of the partial ones for the tiles to reconfigure, and programming the target FPGA.

The hardware-software DSE framework, depicted in Figure 4.6, takes as its inputs the accelerator tiles for the application to execute on the SoC, the properly configured SoC, including its topology and the placement of the various non-accelerator tiles, and a set of performance requirements for the accelerated applications as well as constraints for for the FPGA resource utilization. The DSE flow's key phases are described below.

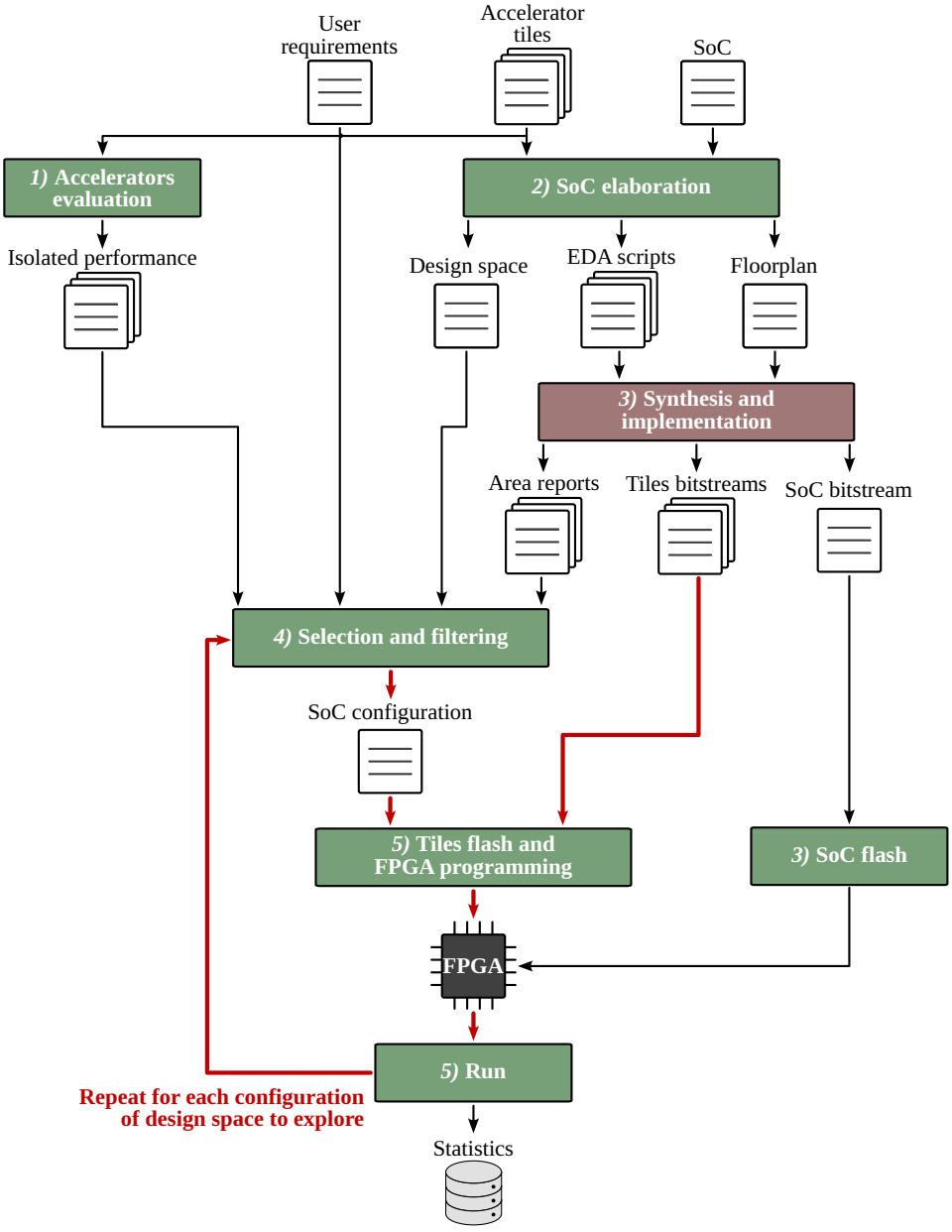


Figure 4.6: *Detailed flow of the Omega framework for the DSE of FPGA-based heterogeneous multi-core SoCs.*

Preliminary Accelerator Evaluation

The accelerator tiles, that may also include multiple accelerators of the same type according to a design-time configurable parameter, are first evaluated by instantiating SoCs in which they are the sole accelerator and they are placed next to the memory tile, thus delivering the maximum achievable performance for that single computing tile. The obtained performance metrics will be used later to discard unfeasible SoC configurations and skip their flashing on the device and board execution.

Omega SoC Elaboration

The elaboration of the target SoC follows such preliminary evaluation, generating the files employed by the EDA tool to carry out synthesis and implementation of the accelerator tiles and the rest of the SoC. Such files include the floorplan constraints that map the reconfigurable regions on the device, the scripts that drive the EDA tool across the DPR flow to generate the bitstreams of the accelerator tiles, and the list of SoC configurations.

Synthesis and Implementation

The synthesis, implementation, and bitstream generation performed by the EDA tool produce a SoC bitstream containing empty tiles in place of the accelerators and the partial bitstream of each accelerator tile for each reconfigurable region. The number of partial bitstreams corresponds to the product of the number of accelerator designs, the maximum number of accelerators instantiated in each accelerator tile, and the number of reconfigurable regions in the Omega SoC. Even though such large number of partial bitstreams causes the implementation process to last for several hours even for medium-sized designs, such activities are carried out just once at the start of the DSE flow and the subsequent SoC instantiations become dramatically faster by leveraging the DPR capability of the target FPGA chip. The static SoC bitstream is then flashed onto the device, providing the baseline design that will be adjusted to each specific SoC configuration by loading the partial bitstreams for each tile.

The implementation phase generates area reports for each tile, which can be used with the isolation performance results to determine the feasibility of SoC configurations prior to testing them, and for the rest of the SoC. Such area results can be extended to a functionally-equivalent SoC implemented without DPR capabilities, since the DPR infrastructure accounts for minimal area overhead and any differences in the area results should be accounted

only to optimizations across different functional blocks of the EDA tool, which are however highly limited due to the SoC's tile-based architecture.

Selection and Filtering

Each DSE iteration selects a new tile configuration and compares its FPGA resource utilization from tile implementation reports and maximum achievable performance from isolated evaluation against the requirements and constraints defined by the system designer. If the resulting area and performance does not satisfy the latter, the next tiles configuration is then selected, otherwise the current one is processed to be instantiated on the prototype FPGA board, flashing the corresponding partial bitstreams on the FPGA.

Flash, Programming and Run

The host programs the CPU tile of the SoC with the testbench executable, the program is launched, and the execution statistics are sent back to the host and collected in a database. Such sequence of steps is repeated for all the possible combinations of accelerator tiles for the required applications, including different levels of replication and positions of the tiles in the NoC interconnect.

Once the design space given by the desired set of applications has been fully explored, the globally optimal SoC instance can be identified with 100% confidence according to the database of performance and area reports. We remark that the proposed flow does not leverage any model or heuristic to explore a portion of the design space and possibly identify a local optimal point, but instead the Omega framework provides real quality metrics for each single point in the design space.

CHAPTER 5

Optimization policies

The Omega framework, presented in Chapter 4, provides the technological foundation for generating diverse SoC configurations and exploring the corresponding design space. Although its dynamic reconfigurability enables faster evaluation of configurations, exhaustively exploring large design spaces–often containing hundreds of thousands of configurations–remains infeasible within reasonable time constraints. Therefore, the Omega framework must be coupled with optimization methodologies that can effectively guide exploration by identifying configurations with the highest likelihood of being optimal, thereby eliminating the need for exhaustive testing.

In this research, we addressed the problem by dividing it into two complementary subproblems. The first concerns HW/SW co-design, which focuses on determining which parts of an application should be executed in software and which should be accelerated in hardware. This optimization is crucial for efficiently exploiting hardware resources, ensuring that only those application functions with the greatest performance benefit are offloaded to hardware accelerators.

Once the relevant accelerators have been identified, the second problem arises: accelerator allocation. The complexity of this task stems from the interactions among multiple accelerators sharing common resources (e.g.,

interconnections, memory), which makes it impossible to derive a closed-form expression for predicting accelerator performance under high-traffic conditions. To address this challenge, we adopted a novel approach based on machine learning techniques.

The remainder of this chapter is organized into two sections. Section 5.1 presents the application-level HW/SW co-design methodology for optimizing application execution patterns on heterogeneous platforms. Section 5.2 introduces FARMER, a machine-learning-based iterative strategy for identifying the optimal allocation of accelerators.

5.1 Accelerator-level optimization

This section presents a HW/SW co-design methodology aimed at heterogeneous SoC architectures that combines CPUs with programmable logic, which has the goal of identifying the optimal allocation of the computing kernels that compose an application.

The proposed methodology is composed of three steps. After identifying which operations to instantiate in hardware, we generate and optimize the corresponding accelerators through HLS, and finally we integrate them in the heterogeneous SoC and make use of their functionality in the software code.

5.1.1 Performance profiling

The first step foresees identifying which portions of the target application to accelerate in hardware and which to execute in software. Intuitively, we aim to find a small number of operations that occupy a large share of execution time and implement those operations in hardware to speed up their computation.

To achieve this, we execute the target application several times on the main SoC CPU, and gather the average execution time of all the main operations. Generally speaking, most applications are dominated by a minimal number of operations, often composed by heavy arithmetical computation and/or deep nested loops. On the contrary, control-flow operations have a much lower impact on the global execution profile and are much more differentiated in their behavior, making their acceleration in hardware impractical and less useful.

5.1.2 High-level synthesis of hardware accelerators

Once the application profiling is completed, a list of all the candidate operations for hardware acceleration is available. In order to rapidly deploy and prototype the accelerators, we use HLS, that allows to generate an hardware accelerator starting from the same software code employed for the application profiling. Indeed, some corrections must be applied to the code in order to make it suitable for HLS (e.g., replacing unbounded arrays with bounded arrays or restructuring recursive functions).

After this preliminary step, we proceed by configuring the area-performance trade-off of the computing kernel using HLS pragmas. These directives allows to easily modify the hardware implementation of the operations, allowing for different levels of parallelization and pipelining as well as mapping the software code on different hardware primitives. This represents a streamlined and systematic approach for identifying Pareto-optimal configurations, as it requires only the adjustment of a set of parameters.

5.1.3 Integration of hardware accelerators in heterogeneous SoC

Indeed, not all the accelerators generated in this way can be integrated in the SoC. The choice of the accelerators that should be actually implemented is primarily guided by the available FPGA resources. As a general rule, integrating a new hardware accelerator will subtract precious resources to the other accelerators, that will be forced to change their Pareto-optimal configuration by reducing their area and, consequently, their performance. For this reason, instantiating too many accelerators would cause a negative effect on performance.

The chosen accelerators are then integrated in the Omega SoC described in Chapter 4. The use of a standard protocol (AXI-Stream) for the accelerator sockets allows for a fast instantiation of modules generated with HLS, whose output connections can be easily configured to adopt standard protocols with the use of specific HLS pragmas. The CPU can easily access and execute these accelerators through a memory-mapped interface, meaning that the software and hardware sections of the application can work together in a straightforward way.

5.2 FARMER: an online-learning optimization methodology

The problem of selecting the optimal FPGA configuration, defined as the one that meets service-level agreements (SLAs) while minimizing area usage to reduce provider costs, presents two major challenges. First, the

design space increases exponentially with the number of applications and the available FPGA resources, making exhaustive exploration impractical. Second, accurately modeling application performance in a multi-accelerator setting is inherently difficult. Performance is strongly influenced by memory access latency, which depends on the number and timing of memory requests from other concurrently executing accelerators.

These two challenges fueled the development of several heuristics to speed up the identification of an optimized configuration [29, 31, 49, 94, 111]. However, memory latency is significantly affected by the underlying computing architecture, necessitating the use of heuristics tailored to the specific problem, which prove ineffective when addressing the issue at a larger scale. Moreover, optimization strategies are often based on data extracted from cycle-accurate simulators [22], but their low accuracy limits the quality of the selected solution with respect to on-board testing.

For this reason we propose FARMER, an online learning methodology to consolidate large FPGAs with a set of hardware accelerators, each of them associated with a specific SLA. FARMER takes in input the RTL description of the applications, the SLA for each application, and the available resources on the FPGA, and produces a feasible configuration that consolidates the applications on the FPGA.

The use of a learning-based strategy allows to overcome the two major challenges in the optimization of large SoCs. First, it removes the need of exploring the whole design space, since ML modeling can predict the performance of the accelerators even in traffic-intensive scenarios without testing each configuration. Second, to perform ML prediction it is not necessary to specifically model the target architecture. In fact, ML can automatically extract a model of the system from a dataset of execution statistics. This makes an learning based approach much more generalizable than a traditional heuristics, since the same methodology can be applied to different architectures by just providing the correct data to the model.

FARMER consists of three tightly connected components represented in Figure 5.1. For each application, the methodology instantiates an ML regressor to predict throughput. Notably, the FARMER flow is generic and is not tailored to any specific ML model. Using the predictions performed by the models, and leveraging the principle of *optimism in the face of uncertainty*, the FC-MAB component has the task of determining the next configuration to prototype, aiming to *(i)* identify a feasible mapping of applications onto the FPGA, and *(ii)* maximize the utility of the extracted data that will be used to improve the ML predictors. Last, the in-circuit exploration flow implements the hardware infrastructure to automatically

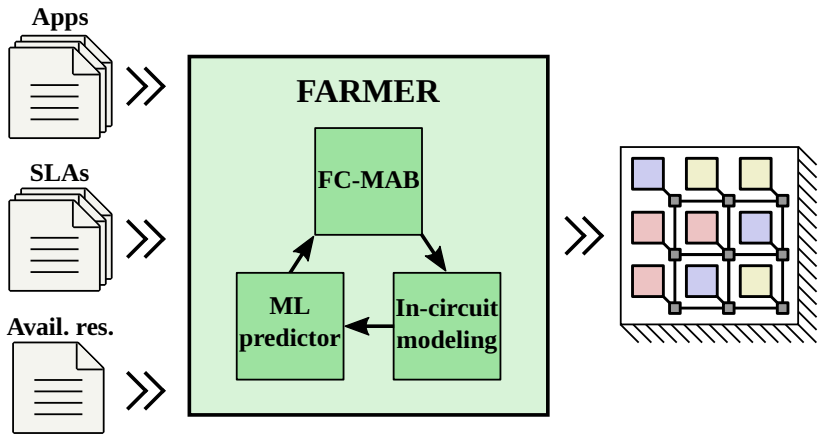


Figure 5.1: *Overview of FARMER.*

prototype the configuration selected by the FC-MAB, execute it on the FPGA, and feed the collected results to the GP predictors to improve the accuracy of the subsequent estimates.

The rest of this section is organized into four parts. Section 5.2.1 defines the optimization problem at the core of the proposed FPGA consolidation strategy. Section 5.2.2 details the formulation of the ML regressor capable of modeling the throughput of an accelerator. Section 5.2.3 discusses the online decision rule that guides the DSE. Last, Section 5.2.4 describes the in-circuit exploration flow.

5.2.1 Optimization problem formulation

Let us consider a set of N applications $P = \{p_i\}_{i=1}^{N}$, each associated with a SLA specifying a minimum throughput requirement $\Theta_i \in \mathbb{R}^+$.

Without loss of generality, we assume the FPGA logic is partitioned into T tiles, with $T > N$ to ensure a sufficient number of slots for accelerator deployment. Each tile t_j, for $j \in \{1, \ldots, T\}$, can be configured to host one of the following:

1. an accelerator implementing an application p_i,
2. a memory channel, or
3. left unused (empty).

To meet SLA requirements, the same accelerator can be instantiated in multiple tiles to increase the total throughput of its corresponding application. Furthermore, each accelerator instance can be configured with a specific

parallelism level $r \in \{0, \ldots, R_{\max}\}$, where $R_{\max}$ denotes the maximum level of parallelism permitted on a single tile, and $r = 0$ implies the tile is unused. It is important to note that $R_{\max}$ is primarily determined by the available resources of a tile, since each one includes $1/T$ of the total FPGA resources destined to the workloads acceleration.

Each accelerator $\mathbf{x}_j$ is thus characterized by a parallelism level r_j, by a corresponding application $s_j \in P$, and by an area a_j, which represents the amount of logic resources consumed by the accelerator.

A configuration is defined as $c = (M, \mathcal{A})$, where M is the number of memory channels and $\mathcal{A} = \{\mathbf{x}_j\}_j$ is the set of instantiated accelerators. When a configuration c is evaluated, it returns a throughput vector:

$$\left(\theta_1(\mathbf{x}_1), \ldots, \theta_{|\mathcal{A}|}(\mathbf{x}_{|\mathcal{A}|})\right), \tag{5.1}$$

where $\theta_j(\mathbf{x}_j)$ represents the measured throughput of accelerator $\mathbf{x}_j$, and $|\cdot|$ denotes set cardinality.

The optimization objective is to determine a feasible configuration $c \in C$ that minimizes the overall area consumption while satisfying throughput constraints:

$$\min_{c \in C} \quad \sum_{j=1}^{|\mathcal{A}|} a_j \tag{5.2}$$

$$\text{s.t.} \quad M + |\mathcal{A}| \leq T \tag{5.3}$$

$$\sum_{j : s_j = p_i} \theta_j(\mathbf{x}_j) \geq \Theta_i \quad \forall i \in \{1, \ldots, N\} \tag{5.4}$$

Constraint (5.3) ensures that the total number of allocated tiles, both for memory and accelerators, does not exceed the available FPGA capacity T. Constraint (5.4) guarantees that the cumulative throughput for each application p_i satisfies its corresponding SLA Θ_i. [1]

5.2.2 ML-based regressor

The complex interactions among applications sharing FPGA resources prevents the problem defined in Section 5.2.1 from admitting a closed-form solution. Specifically, the throughput values $\theta_j(\mathbf{x}_j)$ associated with the accelerators $\mathbf{x}_j$ in any given configuration are not known a priori and necessitate the use of a learning-based strategy.

[1] For clarity of presentation, we omit the trivial constraint that requires at least one memory channel to be present in every configuration, i.e., $M \geq 1$.

To deliver a lightweight yet effective representation of accelerators, each application is classified as either *compute-bound* or *memory-bound*. This abstraction enables the methodology to remain largely independent of low-level implementation details, while accurately modeling the performance degradation caused by concurrently executing applications.

Each accelerator $\mathbf{x}_j$ is described by the tuple:

$$\mathbf{x}_j = \left(r_j, o_j^{(\text{mb})}, o_j^{(\text{cb})}, s_j, a_j \right), \tag{5.5}$$

where r_j denotes the level of parallelism of the accelerator, $s_j \in P$ identifies the application being implemented, and a_j represents the area required on the FPGA. The terms $o_j^{(\text{mb})}$ and $o_j^{(\text{cb})}$ indicate the number of memory-bound and compute-bound accelerators, respectively, that share the same memory channel with $\mathbf{x}_j$.

The relationship between an accelerator and its corresponding throughput is modeled by the feature vector $\mathbf{z}_j := (r_j, o_j^{(\text{mb})}, o_j^{(\text{cb})})$.

Given a dataset of $k-1$ previously evaluated accelerators $\{\mathbf{x}_h\}_{h=1}^{k-1}$ of the same application type s_j, with associated feature vectors $\{\mathbf{z}_h\}_{h=1}^{k-1}$ and throughput measurements $\{\theta_h(\mathbf{x}_h)\}_{h=1}^{k-1}$, the regression model estimates the expected throughput $\hat{\mu}_{j,k-1}(\mathbf{z})$ for a new accelerator characterized by $\mathbf{z}$. In addition to the predicted mean, the model also provides a standard deviation $\hat{\sigma}_{j,k-1}(\mathbf{z})$, quantifying the uncertainty of the estimate.

While several learning models from the literature are suitable for constructing the predictor described above, selecting the most appropriate one is non-trivial, as it depends on both the structure of the optimization problem and the specific features considered. To this end, we opted for making FARMER agnostic to the choice of the machine learning algorithm. This flexibility enables an experimental comparison of different models, allowing the selection of the most effective approach for the given optimization scenario.

5.2.3 Online decision rule

The role of the FC-MAB component is to evaluate the predictions produced by the ML regressors across the entire configuration space, therefore guiding the progression of the exploration process.

The configuration selection strategy at each iteration k is based on the *optimism in the face of uncertainty* principle. This decision rule enhances the likelihood of identifying promising configurations by favoring those with high potential throughput under uncertainty.

In particular, at each iteration, we solve the optimization problem defined by Equations (5.2)–(5.4), replacing the unknown throughput $\theta_j(\mathbf{x}_j)$ with an optimistic upper bound estimated by the regressor:

$$\hat{u}_{j,k-1}(\mathbf{z}) = \hat{\mu}_{j,k-1}(\mathbf{z}) + b_{j,k-1}\hat{\sigma}_{j,k-1}(\mathbf{z}), \tag{5.6}$$

where $\hat{\mu}_{j,k-1}(\mathbf{z})$ and $\hat{\sigma}_{j,k-1}(\mathbf{z})$ denote the predicted mean and standard deviation of the throughput for the feature vector $\mathbf{z}$, respectively, and $b_{j,k-1} \in \mathbb{R}^+$ is a tunable parameter that controls the level of optimism.

It is worth noting that relying solely on the predicted mean $\hat{\mu}_{j,k-1}(\mathbf{z})$ may hinder convergence to the optimal solution. In contrast, the use of optimistic estimates has been shown to enable asymptotic convergence to the optimum, as demonstrated in classical algorithms such as UCB1 [18] and its GP-based counterpart GPUCB [100].

Algorithm 2 FC-MAB Algorithm

1: Initialize the ML predictors with a configuration c_o providing $\theta_j(\mathbf{x_j})$, with $j \in \{1, \ldots, |\mathcal{A}_o|\}$
2: **for** $i \in \{1, \ldots, I\}$ **do**
3: Compute $\hat{u}_{j,i}(\mathbf{z})$ for each $j \in \{1, \ldots, N\}$ and for each $\mathbf{z}$
4: Select the configuration c_i solving the optimization problem in Equations (5.2)-(5.4) using $\hat{u}_{j,i}(\mathbf{z})$ instead of $\theta_j(\mathbf{x})$
5: Get the throughput $\theta_j(\mathbf{x})$ for each $j \in \{1, \ldots, |\mathcal{A}_i|\}$
6: Update the ML predictors with the newly collected values for z and $\theta_j(\mathbf{x})$
7: **end for**

Algorithm 2 provides the pseudo-code describing the feedback loop implemented by the FC-MAB optimization strategy.

5.2.4 In-circuit exploration flow

Figure 5.2 illustrates the online-learning optimization flow of the FARMER methodology. It takes as input a set of target applications, their SLAs, a set of SoC design constraints (e.g., NoC topology), and the available FPGA resources, and produces an optimized configuration as output. It should be noted that FARMER does not guarantee identification of the absolute optimal configuration, as that would require exhaustive exploration of the design space, which is typically infeasible due to its prohibitive size.

The FARMER flow begins with input elaboration, which generates a comprehensive list of all feasible configurations within the design space (`design space list` in Figure 5.2). This list includes all SoC designs that *(i)* satisfy the user-defined SoC rules, *(ii)* include at least one accelerator for each target application, and *(iii)* exhibit resource utilization compatible

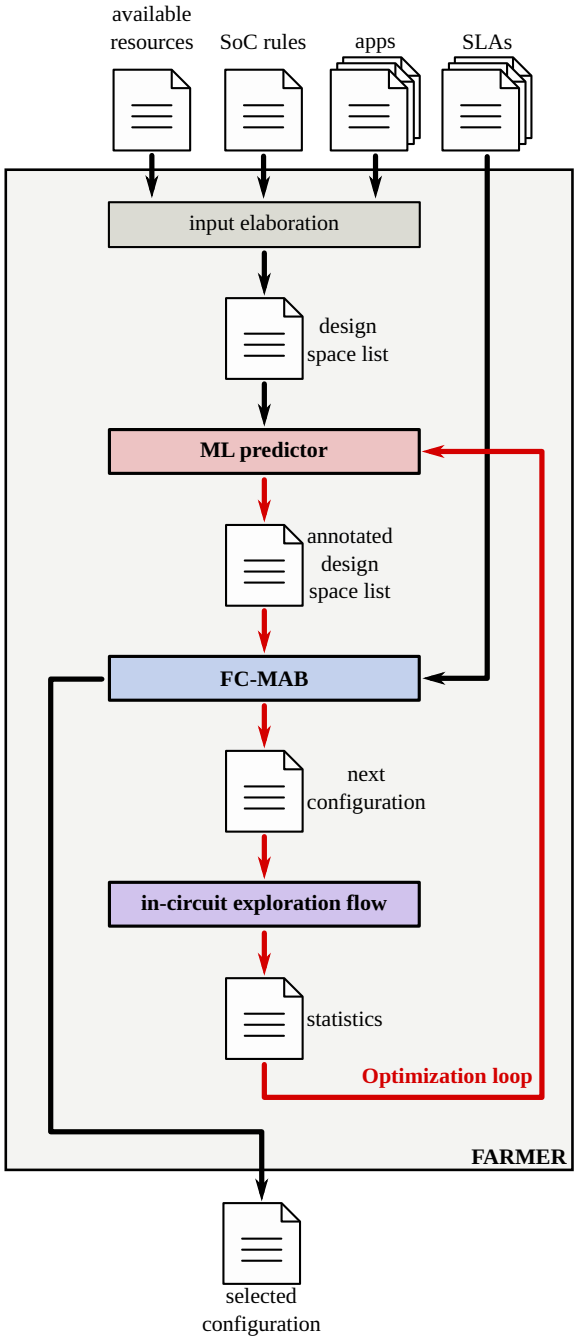


Figure 5.2: *The FARMER online learning pipeline.*

with the selected FPGA device. After this initialization step, FARMER enters the optimization loop, which comprises the three components shown in Figure 5.1: the `ML predictor`, the `FC-MAB` decision rule, and the `in-circuit exploration flow`.

For each application, the `ML predictor` estimates accelerator performance under a given configuration. This process is repeated for every configuration in the design space, resulting in an `annotated design space list`. Based on this annotated information, the `FC-MAB` component selects the next configuration to be tested, guided by the optimistic upper bounds introduced in Section 5.2.3. The quality of each configuration selection directly depends on the accuracy of the ML predictors. At the beginning of the optimization process, FARMER explores configurations almost randomly, sampling broadly across the design space. As the loop iterates and the ML models are refined, the exploration gradually converges toward the regions of the design space with the highest probability of containing near-optimal configurations.

The selected configuration is then evaluated through the `in-circuit exploration flow`, which implements and tests it directly on the FPGA device. The resulting execution statistics are used to retrain the ML predictors, thereby improving their accuracy and guiding subsequent iterations. At the conclusion of the iterative process, FARMER returns the most promising configuration from the explored set—specifically, the one that minimizes area utilization while still meeting the SLA constraints for all applications.

The hardware and software realization of the in-circuit exploration flow is crucial to ensure the practical viability of FARMER in real-world scenarios. Consider, for example, a naive exploration flow that re-implements each configuration from scratch. Given that synthesizing and implementing a medium-sized FPGA design can take tens of minutes on modern datacenter-class CPUs, a complete exploration under such a flow would require several hours or even days—an impractical duration for server consolidation environments, where incoming requests are frequent and delivery time is crucial.

To address this limitation, the Omega DPR-accelerated exploration flow described in Section 4.4 is adopted. This approach enables rapid evaluation of new configurations without requiring full re-implementation. Specifically, the implementation phase is performed only once at the start of the exploration to produce a static bitstream, comprising the NoC and other fixed SoC components (e.g., I/O), along with partial bitstreams corresponding to the reconfigurable accelerator designs. As a result, no synthesis or implemen-

tation is performed within the optimization loop. Testing a configuration simply involves assembling it by programming the FPGA with the static bitstream and the appropriate partial bitstreams, achieving a reduction in exploration time of more than an order of magnitude.

CHAPTER 6

Experimental results

This chapter presents a series of experimental campaigns designed to illustrate the capabilities of the Omega framework introduced in Chapter 4 and to validate the optimization methodologies discussed in Chapter 5. Accordingly, the reported results encompass a broad range of experiments, from area and performance characterization of the various components of the Omega SoC prototyping tool to complete DSEs, demonstrating the effectiveness of the proposed optimization approaches. Importantly, all performance data were collected on real FPGA boards, underscoring the observability offered by the Omega framework. Moreover, the experiments were conducted under diverse hardware and software setups (detailed at the beginning of each section), highlighting the flexibility of the proposed SoC prototyping tool.

The experiments described in this chapter employ several benchmarking applications, which are summarized in Table 6.1. In particular, most of the accelerators were obtained through HLS of applications from the CHStone benchmark suite [40], a collection of applications specifically meant for C-based high-level synthesis. Two accelerators (*aes* and *sha*) are instead high-performance HDL implementations of two cryptographic algorithms, derived from [48] and [50], respectively.

Table 6.1: *A list of all the accelerators employed for benchmarking the SoC functionalities and the optimization techniques presented in this work.*

Name	Description	Impl.	Source
adpcm	Adaptive differential pulse code modulation	HLS	[40]
blowfish	Data encryption standard	HLS	[40]
dfadd	Double-precision floating-point addition	HLS	[40]
dfdiv	Double-precision floating-point division	HLS	[40]
dfmul	Double-precision floating-point multiplication	HLS	[40]
dfsin	Double-precision floating-point sine function	HLS	[40]
gsm	Linear predictive coding analysis of global system for mobile communications	HLS	[40]
mips	Simplified MIPS processor	HLS	[40]
motion	Motion vector decoding of the MPEG-2	HLS	[40]
aes	Advanced encryption standard	HDL	[48]
sha	Secure hash algorithm	HDL	[50]

The remainder of this chapter is structured as follows. Section 6.1 characterizes the Omega framework, with a focus on its DFS and multi-replica architectures. Section 6.2 addresses the memory benchmarking platform, demonstrating its versatility through extensive measurements on a DDR4 FPGA board. Section 6.3 assesses the capability of the DPR mechanism to explore large design spaces by performing exhaustive DSEs and comparing exploration times against conventional methodologies. Section 6.4 introduces a use-case for the HW/SW co-design strategy, presenting the corresponding area and performance results. Section 6.5 evaluates the FARMER online-learning methodology in realistic DSE scenarios, emphasizing its ability to identify feasible configurations while exploring only a small fraction of the design space. Finally, Section 6.6 reports the results of DSEs carried out using the FARMER exploration strategy integrated with the Omega prototyping framework.

6.1 Omega SoC characterization

The experimental evaluation, targeting representative tile-based multi-core heterogeneous SoCs generated by the proposed framework and deployed on a large FPGA chip, demonstrates in the first place the impact on performance and resource utilization of adopting the proposed multi-replica accelerator-tile architecture as well as the flexibility to partition the SoC into any number of frequency islands with or without DFS support and the corresponding area overhead.

Table 6.2: *FPGA resource utilization and throughput of the baseline accelerator tiles and of their 2× and 4× multi-replica instances. Legend:* ***Accel.*** *accelerator,* ***Thr.*** *throughput (MB/s),* ***Incr.*** *average increment compared to baseline accelerator tile.*

Accel.	Replication	LUT	FF	BRAM	DSP	Thr.
adpcm	1×	10899	11720	25	81	1.40
	2×	16455	15158	48	162	2.76
	4×	27313	21780	94	324	5.41
aes	1×	7969	10224	28	9	2.11
	2×	10303	11878	54	18	4.12
	4×	14968	15186	106	36	7.90
blowfish	1×	11596	15373	25	6	2.87
	2×	17856	22466	48	12	5.62
	4×	30044	36395	94	24	10.78
dfadd	1×	11268	11199	2	9	9.22
	2×	16988	14090	2	18	16.88
	4×	28599	19614	2	36	26.06
dfdiv	1×	10179	12244	2	33	2.12
	2×	14574	15920	2	66	4.15
	4×	23866	23274	2	132	7.75
dfmul	1×	8435	10222	2	25	8.70
	2×	11352	12136	2	50	15.07
	4×	17382	15706	2	100	26.06
dfsin	1×	16627	14997	2	52	0.33
	2×	27770	21686	2	104	0.65
	4×	50043	34804	2	208	1.24
gsm	1×	9900	11418	18	62	4.61
	2×	14304	14520	34	124	8.90
	4×	22927	20473	66	248	16.67
mips	1×	7518	9441	5	16	0.50
	2×	9771	10568	8	32	1.00
	4×	13845	12568	14	64	1.99
motion	1×	9063	10485	12	9	0.39
	2×	12263	12368	22	18	0.75
	4×	18705	16135	42	36	1.37
sha	1×	9824	11908	138	15	0.01
	2×	13946	15212	274	30	0.02
	4×	22396	21825	546	60	0.03
Incr.	**2×**	1.44×	1.27×	1.55×	2×	1.92×
	4×	2.31×	1.79×	2.66×	4×	3.58×

Our experimental campaign highlights the capabilities of the proposed agile SoC development framework, including its configurability as well as monitoring and actuation features, through the DSE of a 4-by-4 tile-based SoC that explores the accelerators' performance depending on the clock frequencies of the various islands, the replication factor of the accelerator tiles, their positioning in the NoC mesh, and the amount of traffic on the interconnect.

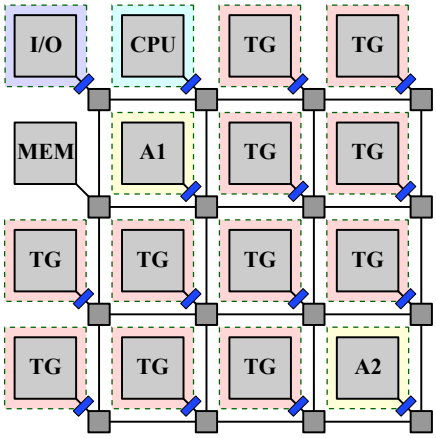
The rest of this section is split into four parts. Section 6.1.1 presents the experimental settings, including the architecture of the SoC instances used throughout our experimental campaign. Section 6.1.2 analyzes the area-performance trade-offs of deploying the proposed multi-replica accelerator tiles, while Section 6.1.3 provides details the area overhead of the proposed frequency island architecture. Finally, Section 6.1.4 demonstrates a DSE on the target SoC architecture highlighting the capability to configure a multitude of parameters and monitor performance-related statistics such as the latencies and the data packets moving across the interconnect.

This section discusses the experimental results considering a representative tiled-based prototype generated with the Omega framework for SoC design. We show the possibility of employing our infrastructure to observe the relationship between the NoC traffic, the operating frequencies, and the performance degradation, thus supporting the design of run-time optimization policies for large FPGA prototypes. Notably, we also consider the distance of the accelerator from the tile-based processor as an additional parameter that influences the performance.

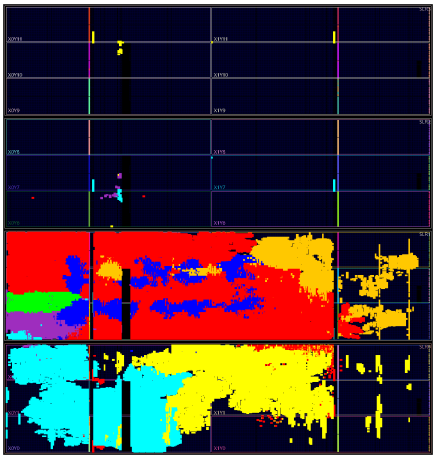
6.1.1 Experimental setup

Hardware setup Our experimental evaluation targeted a Siemens proFPGA quad (*mb-4m-r3*) motherboard [97] equipped with a Virtex-7 2000 FPGA (*fm-xctv2000t-r2*) daughter board [96]. The AMD Virtex-7 2000 (*xc7v2000t-2flg1925-c*) is the largest chip of the Virtex-7 FPGA family and features 1221600 look-up tables (LUTs), 2443200 flip-flops (FFs), 2584 18Kb blocks of block RAM (BRAMs), 2160 DSP elements, and 24 MMCMs.

Software setup The accelerators instantiated in the SoC were obtained through HLS of applications from the CHStone benchmark suite [40]. We employed Cadence Xcelium 20.09 to simulate SoC instances, AMD Vivado 2019.2 for HLS, RTL synthesis and implementation, and bitstream generation, and Siemens proFPGA Builder 2019A-SP2 for FPGA programming, running all the software tools on a host computer with Ubuntu 22.04.3. Area



(a) *Configuration of the SoC instances employed for the experimental evaluation. Colors denote partitioning into five frequency islands. Legend:* ***A1*** *MRA tile next to memory,* ***A2*** *MRA tile far from memory,* ***TG*** *dfadd tile for traffic generation.*



(b) *Floorplan of an instance of the SoC architecture in Figure 6.1a. Legend: NoC in blue, I/O in violet, CPU in cyan, TGs in red, MEM in green, A1 (dfsin) in yellow, A2 (gsm) in orange.*

Figure 6.1: *Architecture and floorplan of a 4×4 SoC employed for the experimental characterization of the Omega framework.*

and timing results refer to the post-implementation netlists of the prototypes. The system-level statistics from the SoC execution are collected from the prototyped SoCs through the proposed run-time monitoring infrastructure that stores them in a dedicated memory before exfiltrating to the host via a USB-to-serial interface. Post-implementation RTL simulations of the SoC were executed in Cadence Xcelium for functional verification purposes.

Use-case architecture Without loss of generality, we evaluated our methodology on 4-by-4 tile-based SoC instances, depicted in Figure 6.1a, that feature a CVA6 CPU tile, a DDR memory one, and an auxiliary tile, while the remaining tiles instantiate HLS-generated CHStone accelerators. *TG* tiles generate traffic in the NoC interconnect and implement *dfadd* accelerators, which were empirically observed to be memory-bound. The accelerator tiles in the *A1* and *A2* positions are the main targets of the experimental DSE, and their position was specifically selected to analyze the accelerators' performance when they are close (*A1*) to and far (*A2*) from memory (*MEM*). The other tiles of the SoC are left unchanged across the DSE.

The SoC is split into five separate frequency islands with support for DFS, denoted by different colors in Figure 6.1a, partitioning it into *i)* the monitored accelerators (*A1* and *A2*), *ii)* the NoC interconnect and mem-

Table 6.3: *FPGA resource utilization of SoC instances with different numbers of frequency islands, disabled or enabled DFS support, and the same computing tiles. Legend:* ***Number*** *of frequency islands,* ***DFS*** *disabled (–) or enabled (✓),* ***Total*** *SoC resource utilization,* ***Increment*** *compared to no-DFS single-island configuration.*

Island config.		Total area (with increment)				
Number	**DFS**	**LUT**	**FF**	**BRAM**	**DSP**	**MMCM**
1	–	264788	242034	1224	582	1
1	✓	265141	242131	1226	582	2
		+353 (+0.1%)	+97 (+0.1%)	+2 (+0.2%)	+0 (+0%)	+1 (+100%)
2	✓	264322	241333	1228	582	4
		-466 (-0.2%)	-701 (-0.3%)	+4 (+0.3%)	+0 (+0%)	+3 (+300%)
3	✓	263955	241436	1230	582	6
		-833 (-0.3%)	-598 (-0.3%)	+6 (+0.5%)	+0 (+0%)	+5 (+500%)
4	✓	264427	241539	1232	582	8
		-361 (-0.1%)	-495 (-0.2%)	+8 (+0.6%)	+0 (+0%)	+7 (+700%)
5	✓	264168	241642	1234	582	10
		-620 (-0.2%)	-392 (-0.2%)	+10 (+0.8%)	+0 (+0%)	+9 (+900%)
Virtex-7 2000		1221600	2443200	2584	2160	24

ory controller (*MEM*), *iii)* the traffic generation (*TG*) cores, *iv)* the CPU core (*CPU*), and *v)* the auxiliary tile (*I/O*). The DFS actuator of the NoC island implements a range of operating frequencies comprised between 10MHz and 100MHz, while the other four clock domains support clock signals with frequencies from 10MHz to 50MHz. The clock frequency ranges include values at 5MHz steps.

Figure 6.1b depicts, as an example, the FPGA floorplan of a 5-island, DFS-enabled SoC instance whose *A1* and *A2* tiles are configured as 4×-replicated *dfsin* and *gsm*, respectively. The whole instance of the SoC occupies, as shown in Table 6.3, 264168 LUTs, 241642 FFs, 1234 BRAMs, 582 DSPs, and 10 MMCMs, which correspond to 22%, 10%, 48%, 27%, and 42% of the Virtex-7 2000 resources, respectively.

6.1.2 Multi-replica accelerator tiles: area-performance analysis

Each baseline HLS-generated accelerator occupies up to 1.4%, 0.6%, 5.3%, and 3.8% of the LUT, FF, BRAM, and DSP resources available on the target FPGA, as listed in Table 6.2, thus motivating the exploration of the possibility to improve their performance by exploiting the proposed multi-replica accelerator-tile architecture.

We explore therefore the instantiation of 2×- and 4×-replication tiles and compare their resource utilization and throughput with the baseline, no-replication versions. The throughput is measured on the accelerator instantiated in the *A1* tile, with clock frequencies of 100MHz and 50MHz for the *NoC-MEM* and *A1*-tile islands, respectively, and with all *TG* tiles disabled, thus providing the best performance achievable by the accelerator due to its proximity to memory and no concurrent requests from the other computing tiles in the SoC.

The increase of LUT, FF, and BRAM utilization is shown to be quite smaller than the replication factor, i.e., 2 and 4, respectively, since such resources are also being employed in the additional logic of the multi-replica accelerator tile. On the contrary, DSP resources end up being around 2 and 4 times as much as in the baseline versions, due to no usage of such resources in the additional logic of the computing tile.

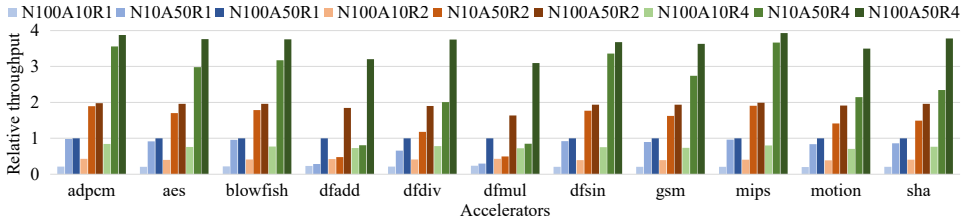
The performance provided by the tile also increases correspondingly with the replication of accelerators, as shown by the throughput columns in Table 6.2. The 2×- and 4×-replication tiles have an average throughput increase of 1.92× and 3.58×, respectively. The larger average throughput improvement compared to the corresponding increase in LUT, FF, and BRAM utilization justifies our multi-replica accelerator-tile solution.

6.1.3 Configurable-DFS frequency islands: area analysis

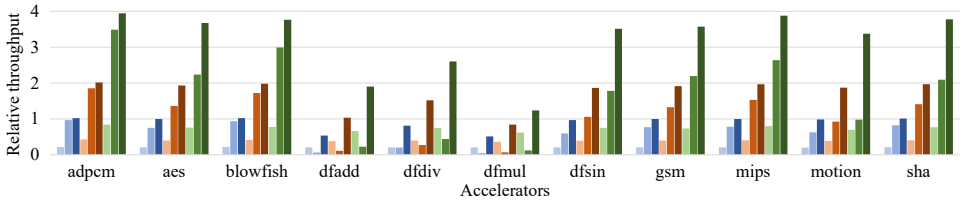
Partitioning the SoC into multiple frequency islands with clock frequencies independently configurable at run-time through their DFS actuators has a limited overhead with respect to the large heterogeneous multi-core processors and the FPGA chips hosting them.

Table 6.3 lists the FPGA resource utilization, in terms of LUTs, FFs, BRAM blocks, DSPs, and MMCMs, of SoC instances partitioned into different numbers of frequency islands and with DFS disabled or enabled, as well as the increment compared to the baseline instance with a single clock domain for the whole SoC and no support for DFS. We explore SoC configurations with a number of frequency islands ranging from 1 to 5, with the 5-island one matching the partitioning depicted in Figure6.1a.

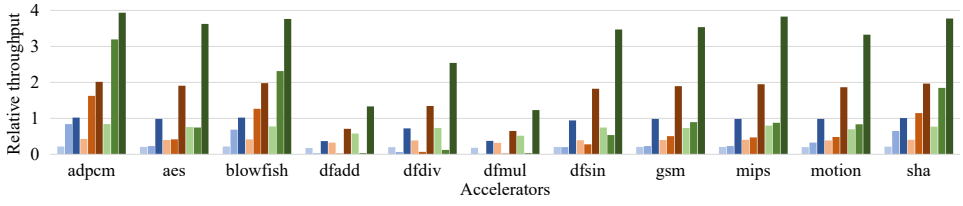
The experimental results highlight very limited variations between the various configurations in terms of LUT and FF resources, which can be traced back merely to differences in the synthesis and place-and-route phases, while the utilization of DSPs is unchanged since such resources are unused in the proposed frequency-island architecture. The BRAM overhead is instead up to 0.8% of the target design, and corresponds to the instantiation, for each island, of two additional 18kb BRAM blocks devoted to storing the frequency configuration parameters for the MMCMs. Finally, MMCMs are shown to be the scarcest resource, due to 2 MMCMs being used in the DFS actuator of each island to guarantee the stability of its generated clock signal. For example, the target AMD Virtex-7 2000 features 24 MMCMs and, thus, supports the partitioning of the SoC into up to 12 configurable-DFS frequency islands.



(a) *A1 accelerator tile, disabled traffic generation.*

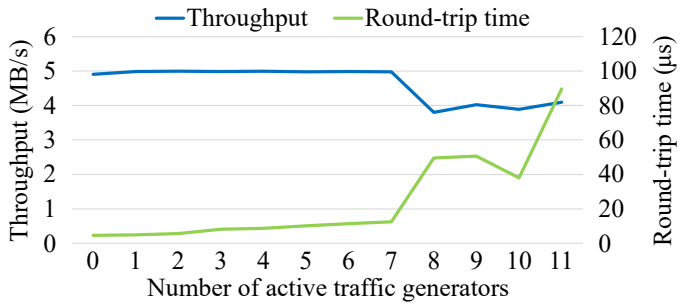


(b) *A1 accelerator tile, enabled traffic generation.*

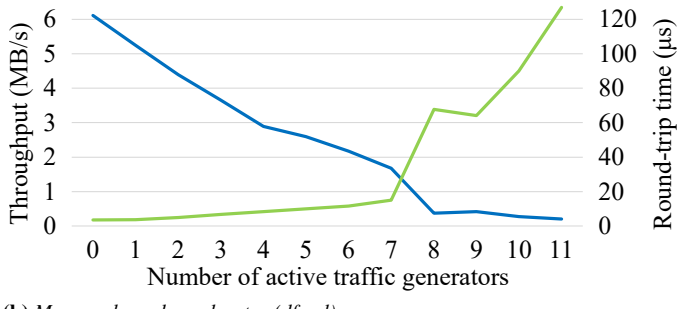


(c) *A2 accelerator tile, enabled traffic generation.*

Figure 6.2: *Throughput of the accelerators with various frequencies for NoC (N in bar labels) and accelerator (A) islands and replication factors (R), in three scenarios differing in accelerator tile positioning and NoC traffic. Values are normalized to N100A50R1 bars in Figure 6.2a.*



(a) *Compute-bound accelerator (adpcm).*



(b) *Memory-bound accelerator (dfmul).*

Figure 6.3: *Throughput and round-trip time of 4×-replication compute- and memory-bound accelerators positioned in the A2 tile at different numbers of active TG cores.*

6.1.4 SoC accelerators: performance analysis

Our experimental DSE aims then to highlight the impact on the performance of the various hardware accelerators given by varying their replication factors in multi-replica accelerator tiles, their position in the NoC, and the clock frequencies of the different islands. Figure 6.2 depicts the throughput of the accelerators when changing such parameters. *NiAjRk* labels for the bars in the charts represent SoC configurations where the NoC interconnect and the analyzed accelerator operate at clock frequencies *i* and *j*, respectively, and the accelerator is replicated *k* times within a single multi-replica accelerator tile.

The three charts represent different scenarios in terms of traffic generated by the *TG* tiles and of positioning of the analyzed accelerator within the 2D mesh NoC. In Figure 6.2a, the accelerator is placed in the *A1* tile position and the traffic generation cores are disabled. In Figure 6.2b, the monitored accelerator is still placed in *A1* but traffic generation is turned on, with *TG* cores operating at a 50MHz clock frequency. In Figure 6.2c, the *TG* cores

are operating as in Figure 6.2b but the accelerator is moved farthest from the memory in *A2*. Each bar in the three charts measures an accelerator's relative throughput by normalizing it to the throughput of the same accelerator in the *N100A50R1* scenario with disabled traffic generation, i.e., the darkest blue columns in Figure 6.2a.

The analysis of the experimental results in Figure 6.2 highlights four main aspects impacting the accelerators' performance:

1. *Replicating accelerators.* Increasing the replication factor in the proposed multi-replica accelerator-tile architecture increases the performance of an accelerator. In all three scenarios, the relative throughput of accelerators in SoC configurations with the same NoC and accelerator clock frequencies increases when augmenting the number of replicas of the accelerator, i.e., *k* in the *NiAjRk* labels.

2. *Overall NoC traffic.* A higher amount of traffic on the NoC interconnect hinders the accelerators' performance. Considering accelerator tiles for the various applications placed in *A1* and keeping the same NoC and accelerator clock frequencies, enabling the *TG* cores as in Figure 6.2b produces lower throughput values compared to the scenario depicted in Figure 6.2a in which the latter are disabled.

3. *Position of tiles in NoC.* Accelerator tiles deployed closer to memory have better performance than the same tiles deployed farther from memory. Moving accelerators for the same application from *A1* to *A2* while maintaining the same islands' frequencies and replication factor decreases the throughput, i.e., the bars in Figure 6.2b are taller than those with the same color and matching label in Figure 6.2c.

4. *Compute- and memory-bound accelerators.* The performance of compute-bound accelerators depends more on the accelerator island clock frequency than on the one of the NoC interconnect, as shown by *N100A10Rk* bars being lower than *N10A50Rk* ones for the *adpcm*, *aes*, *blowfish*, *dfsin*, *gsm*, *mips*, *motion*, and *sha* accelerators. On the contrary, memory-bound accelerators such as *dfadd*, *dfdiv*, and *dfmul* have lower throughput in *N100A10Rk* configurations than in *N10A50Rk* ones in the scenario with disabled traffic generation, i.e., in Figure 6.2a, while with enabled *TG* cores, i.e., in Figure 6.2b, such behavior reverses and *N100A10Rk* bars are taller than *N10A50Rk* ones.

We select *adpcm* and *dfmul* as two representative examples of compute- and memory-bound accelerators, respectively, and analyze the relation be-

tween the amount of traffic in the NoC interconnect and their throughput and average round-trip time.

Figure 6.3 depicts the throughput, in Mb/s, and the average round-trip time, in µs, of the *adpcm* and *dfmul* accelerators, respectively in Figure 6.3a and Figure 6.3b, with a number of active *TG* cores ranging between 0, i.e., when all of them are disabled, and 11, i.e., when all the *TG* cores in the SoC are enabled. The NoC interconnect runs at 10MHz while the accelerators and *TG* cores operate at 50MHz.

The compute-bound *adpcm*, depicted in Figure 6.3a, shows an almost constant throughput between 0 and 7 active *TG* cores, with a corresponding small increase of the average round-trip time. On the contrary, in the same range of X axis values, a small increase of the average round-trip time for the memory-bound *dfmul*, depicted in Figure 6.3b, corresponds instead to a drastic decrease in the throughput of the accelerator.

6.1.5 Memory: traffic analysis

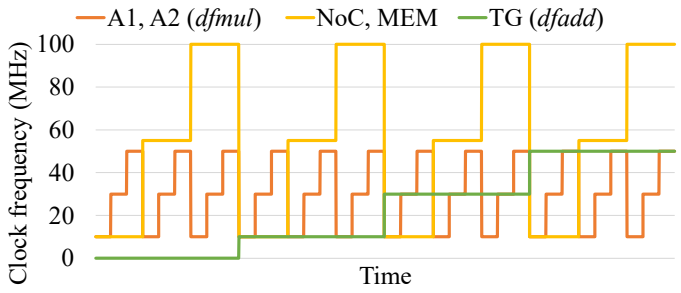
Finally, we show the possibility to evaluate the memory traffic depending on the clock frequencies of the various frequency islands. We consider a specific SoC instance with *A1* and *A2* tiles both instantiating the memory-bound *dfmul* and running concurrently.

Figure 6.4 depicts the temporal evolution of incoming data packets to memory, expressed as millions of packets per second (Mpkt/s) in Figure 6.4b, while varying the frequency islands' clock frequencies, depicted in Figure 6.4a in red for the *A1* and *A2* tiles, yellow for the NoC interconnect and memory controller, and green for the *TG* cores.

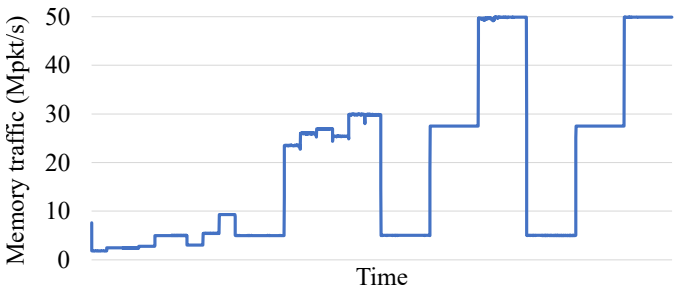
Varying the clock frequency of the *A1* and *A2* tiles between 10MHz, 30MHz, and 50MHz values is shown to have a negligible impact on the memory incoming traffic, while increasing the operating frequency of the *TG* cores drastically increases the pressure on memory when the NoC interconnect and memory controller are also running at a high clock frequency. Such results demonstrate the possibility to analyze the impact of DFS on the local and global traffic of the SoC.

6.2 Memory benchmarking

This section presents a set of experimental results collected with the memory benchmarking platform described in Section 4.5. For this experimental campaign we targeted DDR4 memory chips mounted on an FPGA board. The choice of DDR4 memory technology is motivated by its diffusion in a large variety of computing platforms, from personal computers to data



(a) *Accelerator, NoC, and traffic generators frequency.*



(b) *Packets entering the memory.*

Figure 6.4: *Packets entering the memory tile over time, when varying the frequency of the accelerator under test (dfmul), the frequency of the NoC, and the frequency of the traffic generators (dfadd).*

centers. We note, however, that the proposed framework is general and can be employed with any RAM technology.

The rest of this section is divided in five parts. Section 6.2.1 introduces the DDR SDRAM technology, to provide a clear background on its functioning, while Section 6.2.2 describes the setup adopted for this experimental campaign. Sections 6.2.3, 6.2.4 and 6.2.5 discuss the results collected during this campaign, focusing on the throughput obtained by different communication patterns, on the performance of read-write mixed workloads, and on the performance comparison between memories with different data rates, respectively.

6.2.1 DDR SDRAM background

SDRAM is a widely used type of volatile memory, structured as an array of memory cells. Each memory cell consists of a capacitor and a transistor, which store data as electrical charges. The process of reading from and writing to these cells involves several steps. During a read operation, sense amplifiers detect and amplify the small electrical charge in the cells, con-

verting it into a readable signal. In a write operation, the data is first placed in a row buffer before being transferred into the memory cells. Precharging is a critical step that prepares the memory cells for the next access operation by equalizing the voltages across the bit lines, thus ensuring that the cells are in a known state and ready for accurate reading or writing of data.

DDR is an evolution of the generic SDRAM memory cell (called SDR or single data rate). The main difference is the ability to transfer data two times within the same period, on the rising and falling edge of the clock signal. Aside from generational improvements, this feature effectively doubles the throughput of the preceding SDR cell. DDR4, the fourth generation of DDR SDRAM succeeding DDR3, was introduced to provide higher performance, increased memory capacity, and improved energy efficiency compared to its predecessor. The technology is described by the JEDEC Standard Specification [45].

DDR4 SDRAM is organized into rows and columns, forming a grid structure of memory cells. The memory array is divided into multiple banks, which are further organized into bank groups. This hierarchical structure allows for parallel operations and improved efficiency. By accessing different banks simultaneously, DDR4 can achieve higher throughput and reduce latency compared to earlier memory technologies. DDR4 achieves high data rates through a technique known as double data rate signaling, where data is transferred on both the rising and falling edges of the clock signal. This effectively doubles the data transfer rate without increasing the clock frequency.

The overall performance of DDR4 is influenced by various types of latency, including CAS (Column Address Strobe) latency, which is the delay between the read command and the availability of data, and RAS (Row Address Strobe) to CAS delay, which is the time required to access a specific row before the data can be read or written. Efficient command scheduling and the role of the command and address bus are crucial in coordinating these operations and minimizing delays.

To maintain data integrity, DDR4 memory requires periodic refresh operations. Each memory cell's charge can leak over time, potentially leading to data loss. To prevent this, DDR4 includes a refresh mechanism that periodically rewrites the data in each cell, restoring its charge to the correct level. This process is managed by the memory controller and occurs at regular intervals, ensuring that the data remains intact and accurate. Without these refresh operations, the data stored in DDR4 memory would gradually degrade and become unreliable.

Table 6.4: *Breakdown of the FPGA resource utilization of the DDR4 benchmarking platform instantiated on the proFPGA XCKU115 board.*

Component	LUT	FF	BRAM	DSP
Memory interface	12793	17173	25.5	3
Traffic generator	108	268	0	0
Host controller	70	116	0	0
Single-channel	12975	17559	25.5	3
Dual-channel	25880	35002	51	6
Triple-channel	38785	52445	76.5	9

6.2.2 Experimental setup

The proposed methodology for benchmarking the performance of DDR4 on data-center-class FPGAs with multi-channel capabilities was evaluated through an extensive set of tests on a representative chip in a triple-channel memory configuration. As an example of our solution's flexibility, this manuscript focuses on analyzing the throughput in a multitude of conditions.

The experimental setup consisted of a Siemens proFPGA quad motherboard [97] that mounted an XCKU115 FPGA module [99] and three DDR4 SDRAM memory boards [98]. The proFPGA XCKU115 FPGA module features an AMD Kintex Ultrascale 115 FPGA [10], which can notably host up to three memory controllers [11], while each proFPGA DDR4 SDRAM daughter board provides 2.5GB of DDR4-2400 memory. AMD Vivado ML 2022.1 was employed for the RTL synthesis and implementation, using default optimization directives, while FPGA programming was carried out through Siemens proFPGA Builder 2019A-SP2.

The experiments were carried out at 1600, 1866, 2133, and 2400 MT/s memory data rates, corresponding to JEDEC's DDR4-1600, -1866, -2133, and -2400 standards. In such four operating memory configurations, the PHY and AXI clock domains were accordingly set to operate at 800MHz and 200MHz, 933MHz and 233MHz, 1067MHz and 267MHz, and 1200MHz and 300MHz, respectively, i.e., always maintaining a 4-to-1 ratio. Table 6.4 lists the post-implementation resource utilization of the proposed benchmarking architecture, also breaking it down among the three main components of the architecture, namely, the memory interface, the traffic generator, and the host controller. We remark that one instance of the former two is notably implemented for each memory channel.

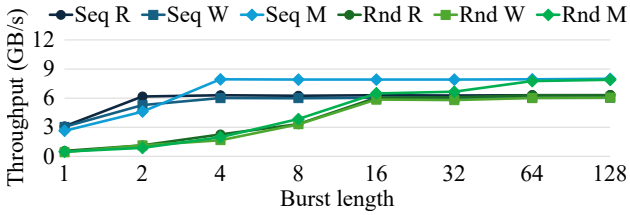
Table 6.5: *Throughput, expressed in GB/s, in the single-channel DDR4-1600 memory configuration.*

Transaction configuration			Addressing mode	
Operation	**Mode**	**Length (#)**	**Sequential**	**Random**
Read	Single		3.08	0.56
	Burst	Short (4)	6.20	2.24
		Medium (32)	6.27	6.08
		Long (128)	6.29	6.30
Write	Single		3.03	0.42
	Burst	Short (4)	6.00	1.66
		Medium (32)	6.03	5.79
		Long (128)	6.04	6.04

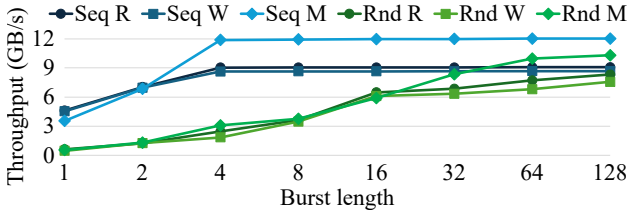
6.2.3 Throughput results

We analyze first the throughput of read and write transactions with sequential and random addressing and with different burst lengths in the DDR4-1600 configuration. The results collected for one memory channel are reported in Table 6.5, that lists the throughput expressed for single transactions and burst ones with lengths of 4, 32, and 128 labeled as short, medium, and long. We remark that the instantiated architecture also supports dual- and triple-channel, thus delivering twice and three times the throughput of the single-channel configuration, respectively.

The throughput drops by up to 5.5× and 7.2× for read and write transactions, respectively, when switching from sequential to random addressing due to the effect of memory precharging, with the highest degradation resulting from single transactions, and sequential read operations achieve a larger throughput than the corresponding write ones. Burst transactions with the smallest burst length show a speedup of around 2× and 4× in sequential and random addressing scenarios, respectively, compared to the corresponding single transactions, due to issuing less address requests to transfer the same amount of data and to the increase in address locality, which also reduces the gap between the two addressing modes. Throughput is notably already close to its maximum value for the short burst with length equal to 4 in sequential addressing mode, while in random addressing mode the performance plateaus at long burst lengths.



(a) *DDR4-1600*



(b) *DDR4-2400*

Figure 6.5: *Throughput of single-channel DDR4-1600 and DDR4-2400 memory configurations. Legend:* ***Seq*** *sequential,* ***Rnd*** *random addressing,* ***R*** *read,* ***W*** *write,* ***M*** *mixed operations.*

6.2.4 Performance breakdown of mixed workloads

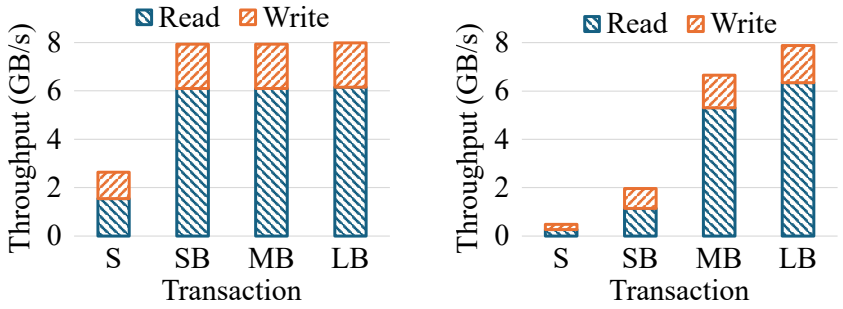
We consider then a mixed workload combining both read and write operations. The proposed platform is indeed able to mix the two types of operations in single transactions as well as in bursts, both with sequential and random addressing. The TG component is able to separately monitor the execution time and number of transactions of the read and write operations in mixed workloads, thus enabling the breakdown of the throughput.

Figure 6.6 depicts, as an example, the throughput breakdown between read and write operations with sequential and random addressing, respectively in Figure 6.6a and Figure 6.6b, with transactions with different burst lengths when the benchmarking platform operates in a DDR4-1600, single-channel memory configuration.

6.2.5 Performance analysis

A fine-grained analysis enables highlighting how throughput behaves in various transaction configurations, including read, write, and mixed operations, sequential and random addressing, and memory data rates. In particular, Figure 6.5 compares the slowest and fastest memory data-rate configurations, i.e., DDR4-1600 and DDR4-2400.

Performance is shown to saturate at different burst lengths when varying the data rate. In the random write case, the DDR4-1600 throughput improves



(a) *Sequential addressing* **(b)** *Random addressing*

Figure 6.6: *Breakdown of the throughput, expressed in GB/s, of mixed read-write transactions in the single-channel DDR4-1600 memory configuration. Legend:* ***S*** *single,* ***SB*** *short (length 4) burst,* ***MB*** *medium (32) burst,* ***LB*** *long (128) burst transactions.*

only by 3% as burst length increases from 16 to 128 in the configuration, while the DDR4-2400 one does not saturate until the highest burst length, with a 24% improvement as burst length increases in the same range. The theoretical 50% throughput improvement given by increasing the memory data rate from 1600 to 2400 MT/s only applies to sequential data transfers under ideal conditions where memory gets accessed in long uninterrupted bursts. While DDR4-2400 transfers data faster, the increase in terms of clock cycles of latency offsets some of those gains, in particular for random accesses, in which every operation involves a delay before the actual data transfer begins.

Sequential accesses benefit most from the bandwidth provided by the increased data rate, while in contrast random accesses are more sensitive to latency than bandwidth, so their performance improvement is significantly less than the theoretical one. The throughput improves by up to 50% and by 44% on average in sequential accesses when switching from DDR4-1600 to DDR4-2400, while in random ones a positive effect is given by increasing the burst length, with the throughput increase being equal to 7% for read bursts with length of 16 and improving up to 32% when their length is equal to 128. The throughput improvement is notably slightly higher for bursts shorter than 16, e.g., of 11% for single transactions and bursts with length equal to 2, due to the greater impact of higher bandwidth compared to higher latency when the throughput has lower values, i.e., of 0.62GB/s and 1.24GB/s for DDR4-2400 random reads.

Mixed workloads are also shown to achieve a higher throughput than solely read and write ones, reaching maximum values with sequential ad-

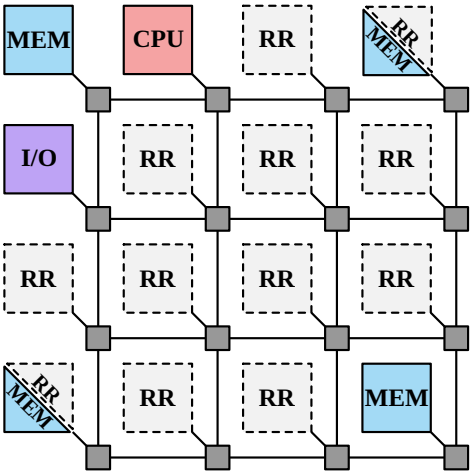


Figure 6.7: *Architecture of the heterogeneous multi-core SoCs to be explored within the experimental campaign.*

dressing of 7.99GB/s and 12.02GB/s for DDR4-1600 and DDR4-2400 data rates, respectively.

6.3 Design space exploration with DPR

The experimental campaign aims to showcase how the Omega framework delivers an effective solution for the exhaustive DSE of FPGA-based heterogeneous multi-core SoCs and for the accurate validation of novel methodologies for DSE.

Section 6.3.1 describes the hardware and software setup of the Omega framework as well as the accelerator designs, SoC configurations, and learning-based DSE approach employed within the experimental campaign, while 6.3.2 discusses the experimental results encompassing the overheads given by applying the Omega methodology and its speedup in the exploration time.

6.3.1 Experimental setup

The Omega framework is run in Ubuntu 22.04.3 LTS on a host PC that features an Intel i5-12400 CPU and a 64GB DDR4 memory. AMD Vivado 2023.2 ML is employed for HLS, synthesis, implementation, bitstream generation, and FPGA programming. The experimental evaluation targets an AMD Alveo U55C PCIe card that hosts an UltraScale+ FPGA chip and

Table 6.6: *FPGA resource utilization, expressed in terms of LUT, FF, BRAM, and DSP resources, and throughput, expressed in MB/s, of the accelerator tiles considered within the experimental campaign.*

Accelerator	Replication	LUT	FF	BRAM	DSP	Throughput
aes	1×	8037	10741	0	1	12.53
	2×	11233	13998	0	2	23.53
dfadd	1×	9899	9664	0	9	7.25
	2×	15228	12099	0	18	14.36
dfdiv	1×	9379	10974	0	33	3.20
	2×	13748	14394	0	66	6.24
dfmul	1×	7783	9105	0	25	7.19
	2×	10641	10979	0	50	14.13
mips	1×	6855	8299	3	16	0.98
	2×	8869	9362	6	32	1.95
motion	1×	8354	9268	10	9	0.77
	2×	11429	11012	20	18	1.45
sha	1×	8850	10574	0	1	273.71
	2×	12850	13915	0	2	322.47

16GB of HBM2. The FPGA features 1303680 LUTs, 2607360 FFs, 4032 18Kb BRAMs, and 9024 DSP elements.

The SoC considered in the experimental campaign features 16 tiles, arranged in a 4×4 topology, and includes a CPU tile, an I/O tile, and either two or four memory tiles. The CPU, I/O, and memory tiles are placed in fixed positions, as depicted in Figure 6.7. Two memory tiles are always placed in the upper-left and lower-right corner, while the upper-right and lower-left tiles may either be memory tiles or accelerator ones. The remaining nodes of the SoC correspond each to a reconfigurable region that supports the dynamic tile swapping capabilities of the Omega framework and that may be either occupied by an accelerator tile or be left empty.

The reconfigurable regions can thus dynamically include tiles with seven different accelerators, namely, the HLS-generated *dfadd*, *dfdiv*, *dfmul*, *mips*, and *motion* from the CHStone benchmark suite and the *aes* [48] and *sha* [50] RTL accelerators that implement the AES and SHA-3 cryptography standards. The accelerator tiles instantiate either one or two identical accelerators. Reconfigurable regions are therefore defined with an amount of resources that can fit any of the accelerator tiles.

Table 6.7: *Applications to be accelerated by the use-case SoCs considered in the experimental evaluation and corresponding performance requirements, expressed in MB/s. – denotes applications that are not required to be supported by SoCs.*

	Design Space		Performance Requirements (MB/s)						
SoC	**Total**	**Filtered**	**aes**	**dfadd**	**dfdiv**	**dfmul**	**mips**	**motion**	**sha**
SoC_1	18578	6389	30	20	–	–	–	–	600
SoC_2		6652	55	12	–	–	–	–	400
SoC_3		5525	40	10	–	–	–	2.5	–
SoC_4		6858	–	25	8	12	–	–	–
SoC_5	79740	7192	28	18	–	–	–	2	550
SoC_6		4986	28	–	–	–	3.2	2	550
SoC_7		5860	–	40	5	20	–	1.5	–
SoC_8		4424	55	–	–	32	1.5	–	250
SoC_9	221055	5264	26	–	7	15	–	2	300
SoC_{10}		4591	15	9	–	9	6	–	300
SoC_{11}		9849	–	38	4	9	1.2	1	–
SoC_{12}		10229	15	9	10	22	–	1	–

Table 6.6 lists the FPGA resource utilization and throughput of the accelerator tiles. Resource utilization is obtained from their post-implementation netlist, while throughput is measured by instantiating just one accelerator tile next to a memory one and no other accelerator tiles in the SoC to avoid any contention of the interconnect and memory resources, thus producing the highest performance achievable by the accelerator tile. The performance statistics are collected from the prototyped SoCs through the run-time monitoring infrastructure and sent to the host via a USB-to-serial interface.

Use-Case SoCs The experimental campaign considers twelve SoCs that differ with each other in the applications to accelerate and their number and in the throughput requirements for each of them. Such properties are listed in Table 6.7.

The twelve SoCs are equally split between those with three, four, and five application workloads to be accelerated in hardware on the target computing platform. Since the SoC architecture, excluding the accelerator tiles, their placement, and their parallelism, is fixed, SoCs with the same number of applications have an overall design space that contains the same total number of possible configurations. Table 6.7 also includes the number of SoC configurations which are found to be feasible after the preliminary analysis, which allows to filter the points of the design space which are

determined to be for certain unable to deliver the required performance given the best-case throughput of the accelerator, which was previously listed in Table 6.6. Finally, each use-case SoC architecture is characterized by the applications it is tasked with accelerating and by the SLA it must provide. Each row of Table 6.7 lists indeed the throughput required by each corresponding application and expressed in terms of MB/s, with – symbols denoting applications that are not part of the various use cases.

The numerical values selected for the requirements were meant to obtain a representative experimental campaign that included various mixes of applications/accelerators and a large design space even after the filtering phase. The twelve use-case SoCs will be used both to evaluate the drastic speedup given by the Omega framework for the exhaustive DSE of a target computing platform and to showcase the capability of Omega to deliver a ground-truth model for the validation of novel analytical models, heuristics, and ML-based techniques for DSE.

6.3.2 Experimental results

The experimental analysis covers the area and performance overheads given by extending a baseline SoC with the hardware required to dynamically swap the accelerator tiles, in Section 6.3.2, and the speedup for conducting an exhaustive DSE compared to a traditional approach, in Section 6.3.2.

Overhead Analysis We evaluate first the overheads, both in terms of area and performance, introduced by the addition of the capability to dynamically swap the accelerator tiles. Additional hardware is indeed added to the I/O tile and to the accelerator tiles, as previously described in Section 4.4.

On the one hand, the control logic added to the I/O tile to manage the dynamic swap of tiles from the host PC occupies 90 LUTs and 100 FFs, i.e., a very limited fraction of the 8054 LUTs, 9391 FFs, and 64 BRAMs of the whole tile. On the other hand, for each accelerator tile, the buffer and the interface between the former and the accelerator logic placed in the reconfigurable region occupy an aggregated number of 15 and 153 additional LUTs and FFs, respectively. BRAM and DSP resources are instead unused in the additional logic of both the I/O and accelerator tiles. The area overheads are almost negligible with respect to the heterogeneous multi-core SoCs targeted by the proposed methodologies, which occupy amounts of LUT and FF resources in the order of hundreds of thousands. In particular, the additional FFs are used to buffer the I/O signals of the

Table 6.8: *Exhaustive exploration time, expressed in hours, of the state-of-the-art reference approach and of the Omega framework, and speedup provided by applying Omega. * denotes estimated times obtained by the exploration of 500 feasible configurations of each design space.*

	SoA Exhaustive DSE			Omega Exhaustive DSE			
SoC	**Synth. & Impl.**	**FPGA Exec.**	**Total**	**Synth. & Impl.**	**FPGA Exec.**	**Total**	**Speedup**
SoC_1	4155*	198*	4353*	57	72	129	34×
SoC_2	4326*	206*	4532*	57	75	132	34×
SoC_3	3593*	171*	3764*	53	60	114	33×
SoC_4	4460*	212*	4672*	53	84	137	34×
SoC_5	4677*	223*	4900*	71	99	169	29×
SoC_6	3242*	154*	3397*	65	68	132	26×
SoC_7	3811*	181*	3992*	67	80	147	27×
SoC_8	2877*	137*	3014*	62	60	122	25×
SoC_9	3423*	163*	3586*	81	62	144	25×
SoC_{10}	2986*	142*	3128*	79	54	132	24×
SoC_{11}	6405*	305*	6710*	77	135	212	32×
SoC_{12}	6652*	317*	6969*	80	141	221	31×
Average	4217*	201*	4418*	67	83	149	29×

reconfigurable region and corresponds to the totality of wires connected to the accelerator interface.

The timing and performance of the SoC is left unchanged, compared to an equivalent configuration without dynamic tile swap capabilities, i.e., each accelerator tile of a baseline SoC and the corresponding one enhanced with the DPR-related logic of the Omega framework produce the exact same throughput.

Exhaustive Exploration Time We evaluate then the time required to perform an exhaustive DSE of the twelve considered SoC architectures with their corresponding throughput requirements, comparing the time taken by the proposed Omega framework against a traditional approach that does not support dynamically swapping the accelerator tiles and requires therefore synthesizing and implementing each single SoC configuration. We apply the same filtering techniques for both the Omega and the reference approach, discarding the SoC configurations which are guaranteed not to be able to satisfy the required performance thresholds.

The experimental results, listed in Table 6.8, highlight a reduction in the exploration time of 29× on average and ranging between 24× and 34×. Exhaustive DSEs that would take between 4 and 10 months with the reference traditional methodology are indeed drastically sped up to require from five to nine days by applying the Omega framework.

The exploration times of the traditional approach notably refer to estimations obtained after running 500 iterations of the DSE, due to their excessive length. In particular, for each of the twelve use-case SoCs, we computed the average time required by a DSE iteration from the 500 actually executed ones and subsequently obtained the estimated exhaustive exploration time by multiplying such average value by the number of configurations to be evaluated after filtering.

Table 6.8 further breaks down the exploration time into two distinct phases, namely, *(i)* synthesis and implementation, and *(ii)* FPGA execution. For the reference approach, the former refers to the synthesis, implementation, and bitstream generation for each configuration, while the latter includes flashing the bitstream, programming the soft-core CPU, and executing the testing workload, which in our experiments has a fixed duration of 10 seconds. The synthesis and implementation phase takes a much larger share of time than the FPGA execution one, as previously remarked in Section 4.7, with an average of more than 95% of the total exploration time.

The synthesis and implementation phase of the Omega framework refers instead to the initial generation of the static and partial bitstreams at the beginning of the DSE. In particular, it corresponds to the implementation of two distinct designs, i.e., one with two memory tiles and another with four memory tiles, since it is not possible to dynamically transition between SoCs with a different number of memory tiles, which are not reconfigurable. Albeit taking longer than a single monolithic implementation, Omega's synthesis and implementation phase takes more than 63× less than the corresponding traditional version due to being executed only once, thus ending up taking even less than the FPGA execution phase, whose much shorter operations are however repeated for each iterations of the DSE. The results also highlight how Omega's FPGA execution itself is 2.5× faster per DSE iteration than the state-of-the-art approach, since flashing a full bitstream in the latter case takes much longer than only loading few smaller partial bitstreams on the FPGA as in Omega, which further reduces the time required for an exhaustive exploration.

6.4 HW/SW co-design results

In this section, we apply the HW/SW co-design strategy presented in Section 5.1 to a real use case, in order to highlight its benefits. The target application is BIKE [14], a post-quantum key encapsulation mechanism (KEM) that employs quasi-cyclic moderate-density parity-check (QC-MDPC) codes in a variant of the Neiderreiter scheme and that has reached the fourth round in the post-quantum cryptography standardization process led by the USA's National Institute of Standards and Technology (NIST) [78].

6.4.1 BIKE HW/SW co-design

The BIKE KEM can be split into three primitives. Key generation produces a public-private key pair, encapsulation generates a shared secret and encrypts it with the public key, and decapsulation retrieves the shared secret with the private key from the exchanged ciphertext. Due to its QC-MDPC code-based nature, BIKE makes wide use of binary polynomial arithmetic operations, preeminently multiplication, exponentiation, and inversion, and of the bit-flipping decoding procedure, in particular the Black-Gray-Flip decoding variant.

In the following, we will detail the application of the three-step methodology discussed in Section 5.1. After identifying which operations to instantiate in hardware, we generate and optimize the corresponding accelerators through HLS, and finally we integrate them in the heterogeneous SoC and make use of their functionality in the software code.

Performance profiling The first step foresees identifying which portions of BIKE to accelerate in hardware and which to execute in software. Intuitively, we aim to find a small number of operations that occupy a large share of execution time and implement those operations in hardware to speed up their computation.

Table 6.9 depicts the performance profile of the execution of the BIKE reference C99 software [14] on an *ARM Cortex-A9* CPU. Notably, bit-flipping decoding and binary polynomial inversion occupy up to 57.0% and 41.1% of the execution time of the whole KEM, respectively. At a finer granularity, binary polynomial inversion is computed as an iterative algorithm composed of binary polynomial multiplications and exponentiations and is primarily dominated by the former. A binary polynomial multiplication also appears in the encapsulation primitive. Cumulatively, bit-flipping decoding and binary polynomial multiplications take 86.9% and 91.0% of the total execution

Table 6.9: *Breakdown of the execution times of BIKE, in milliseconds and percentages, on a 32-bit ARM CPU. Multiplication and decoding operations are highlighted in **bold**.*

KEM primitive	Operation	NIST security level AES-128 [ms]	AES-128 [%]	AES-192 [ms]	AES-192 [%]
Keygen	SHAKE-based PRNG	0.9	0.1%	1.2	0.1%
	Inversion	319.1	39.3%	883.0	41.1%
	→ **Multiplication**	**217.1**	**26.7%**	**696.3**	**32.4%**
	→ Exponentiation	101.9	12.6%	186.7	8.7%
	Multiplication	**12.8**	**1.6%**	**36.6**	**1.7%**
		332.3	40.9%	920.9	42.8%
Encaps	SHAKE-based PRNG	0.9	0.1%	1.2	0.1%
	Multiplication	**12.8**	**1.6%**	**37.2**	**1.7%**
	SHA-3 hash function	0.6	0.1%	1.1	0.1%
	SHA-3 hash function	0.2	<0.1%	0.3	<0.1%
		14.7	1.8%	40.9	1.9%
Decaps	**Decoding**	**463.1**	**57.0%**	**1185.6**	**55.1%**
	SHA-3 hash function	0.6	0.1%	1.1	0.1%
	SHAKE-based PRNG	0.9	0.1%	1.2	0.1%
	SHA-3 hash function	0.2	<0.1%	0.3	<0.1%
		464.8	57.2%	1188.3	55.3%
	Multiplication+Decoding	**705.7**	**86.9%**	**1955.7**	**91.0%**
	Total	812.0	100%	2150.1	100%

time considering AES-128- and AES-192-equivalent security instances of the BIKE KEM, respectively.

Accelerators synthesis The large fraction of BIKE's execution time covered by Black-Gray-Flip decoding and binary polynomial multiplication operations motivates the design choice to accelerate them through HLS-generated accelerators while keeping the computation of the rest of the KEM in software on the CPU.

Binary polynomial multiplication operates on factors with sizes in the order of tens of thousands of bits. The design of the binary polynomial multiplier starts from the C code of a multiplication procedure that leverages the Karatsuba and Comba multiplication algorithms. The Karatsuba multiplication formula is recursively applied a configurable number of times, and, at the innermost level, it makes use of Comba multiplication to compute its partial products. Finally, the Comba multiplier executes its partial products between word-size operands through a combinational multiplier. Configuring the number of Karatsuba recursions enables exploiting the area-performance trade-off. Moreover, unrolling directives in the Comba

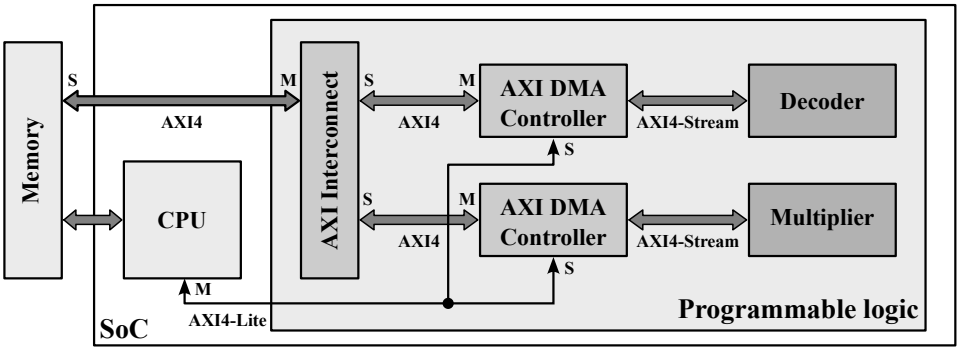


Figure 6.8: *Top view of the heterogeneous SoC architecture. Legend:* ***M*** *AXI master,* ***S*** *AXI slave.*

multiplier allow for reducing latency by setting the length of the combinational multiplier's operands. On the area side, we exploit storage binding and array partitioning HLS directives to use the available resources efficiently, storing the large polynomials to BRAM while reserving flip-flops for variables with sizes up to a few hundred bits.

Black-Gray-Flip decoding is an iterative algorithm that takes a syndrome and the private key as inputs and retrieves an error vector. The complexity of decoding is given by the large size of the code, which is in the order of tens of thousands of bits. The decoding algorithm itself contains multiplications between polynomials. Part of those operations are performed in the Galois field of binary polynomials, while others treat the coefficient of those polynomials as natural arithmetic integer values. On the performance side, we optimize the decoding latency by implementing such multiplications through the same hybrid Karatsuba-Comba approach detailed in the previous paragraph. On the area side, the amount of used FPGA resources is reduced by applying storage binding and array partitioning directives.

Accelerators integration Depending on the amount of available FPGA resources, the accelerators to be implemented in the programmable logic of the SoC are selected and parameterized to maximize the performance of the BIKE execution, i.e., minimize the aggregate execution time of the three KEM primitives.

Hardware accelerators can be integrated within a heterogeneous SoC that features a CPU and programmable logic as depicted in Figure 6.8, which shows an example where the FPGA part instantiates both HLS-generated accelerators for binary polynomial multiplication and Black-Gray-Flip decoding.

Each accelerator exposes an AXI4-Stream interface, which connects it to a dedicated DMA controller that provides high-bandwidth access to the DDR memory. DMA controllers are connected to memory through an AXI4 interface, while the CPU accesses their initialization, status, and management registers through AXI4-Lite.

6.4.2 Experimental setup

Zynq-7000 chips pack a 32-bit dual-core *ARM Cortex-A9* processor, running at a 667MHz clock frequency, connected to an external 512MB DDR3 memory and coupled with *Artix-7*-class FPGA programmable logic. We target the *Zynq-7010* SoC, i.e., the smallest *Zynq-7000* chip, whose FPGA part features 17600 LUT, 35200 FF, 80 DSP, and 120 BRAM resources, and the larger *Zynq-7015* one, with 46200 LUT, 92400 FF, 160 DSP, and 190 BRAM resources. *AMD* tools are employed across the whole workflow, including the *Petalinux 2022.1* OS running on the *Zynq-7000* chips, *Vitis HLS 2022.1* for HLS, *Vivado ML 2022.1* for RTL synthesis and implementation, and *Vitis 2022.1* for programming.

The full AXI4 interface connecting the AXI interconnect module to memory is mapped on the HP0 high-performance dedicated bus, while the AXI4-Lite channel between the AXI DMA controller modules and the CPU is mapped on the GP0 general-purpose bus. The accelerators, interconnect, and DMA controllers are implemented at a 100MHz clock frequency.

The executed software, enhanced by calls to the dedicated hardware accelerators, is the reference C99 implementation from the official BIKE NIST submission [14], which was also employed as the initial source code within the HLS process.

We evaluate the proposed architecture in comparison with a reference implementation of the BIKE KEM, which works at primitive-level granularity through either HLS-generated accelerators or the CPU, and the state-of-the-art software execution of BIKE on the CPU of *Zynq-7000* chips [14].

The FPGA resource utilization of our solutions is obtained after RTL implementation, while the execution times are measured at run time, using the *clock_gettime* function to collect the CPU time at the beginning and the end of the software program or portions of the latter.

6.4.3 Area results

Table 6.10 details the utilization of the FPGA resources of the proposed solutions and compares it to the reference ones, listing the occupied resources

Table 6.10: *Breakdown of the FPGA resource utilization.* ***Our*** *totals include the additional logic to integrate the accelerators in the SoC.*

Design (Target chip)	KEM prim. or op.	NIST security level AES-128 LUT	FF	DSP	BRAM	AES-192 LUT	FF	DSP	BRAM
Ref. HW/SW (Zynq-7010)	Keygen	13567	11621	0	40	N/A	N/A	N/A	N/A
		13567	11621	0	40				
Ref. HW/SW (Zynq-7015)	Decaps	37160	38118	35	90	N/A	N/A	N/A	N/A
		37160	38118	35	90				
Ref. HW/SW (Zynq-7020)	Keygen	13567	11621	0	40	N/A	N/A	N/A	N/A
	Decaps	37160	38118	35	90				
		50727	49739	35	130				
Our (Zynq-7010)	Decod.	12348	11897	14	76	14988	11165	14	125
		12986	11977	14	76	15616	12250	14	125
Our (Zynq-7015)	Mult.	17967	16491	0	36	18760	13691	0	40
	Decod.	24165	21740	14	88	25134	19095	14	133
		42777	39317	14	124	44529	33877	14	173

by breaking them down in the three KEM primitives or the HLS-instantiated operations.

Our designs vary to fit within the resources available on the different chips. The *Zynq-7010* instances implement only the decoder, chosen over the multiplier due to its largest reduction effect on the overall execution time. On the contrary, the *Zynq-7015* instances implement both decoding and multiplication accelerators, as in the architecture depicted in Figure 6.8. The parameters of the HLS designs also differ from each other. Less parallelism is exploited on the smaller *Zynq-7010* instances compared to the larger *Zynq-7015* ones. Moreover, AES-128- and AES-192-equivalent security instances on *Zynq-7015* implement a different number of Karatsuba recursions in the multiplier, five and four, respectively.

Our *Zynq-7015* design occupies a similar number of LUTs and FFs and less DSPs than the reference *Zynq-7015* one, as well as -16% LUT, -21% FF, -60% DSP, and -5% BRAM resources compared to the reference *Zynq-7020* one. Both our *Zynq-7010* and *-7015* instances use more BRAM resources than the corresponding reference ones, thus exploiting more effectively the memories provided by the heterogeneous chips.

6.4.4 Performance results

Performance is measured as the execution time of BIKE, i.e., the aggregate execution time of the three KEM primitives.

Table 6.11: *Breakdown of the execution times, reported in milliseconds, at primitive granularity for the BIKE KEM.*

Design (Target chip)	**KEM primitive**	**NIST security level AES-128**	**AES-192**
Ref. SW [14] (Zynq-7010/15/20)	Keygen	332	921
	Encaps	15	41
	Decaps	465	1 188
	Total	812	2 150
Ref. HW/SW (Zynq-7010)	Keygen	138	N/A
	Encaps	15	
	Decaps	465	
	Total	617	
Ref. HW/SW (Zynq-7015)	Keygen	332	N/A
	Encaps	15	
	Decaps	135	
	Total	482	
Ref. HW/SW (Zynq-7020)	Keygen	138	N/A
	Encaps	15	
	Decaps	135	
	Total	288	
Our (Zynq-7010)	Keygen	332	921
	Encaps	15	41
	Decaps	138	484
	Total	485	1 446
Our (Zynq-7015)	Keygen	145	423
	Encaps	5	12
	Decaps	93	311
	Total	243	747

Table 6.11 compares the proposed solution to the reference works, listing, for each design and each considered security level of BIKE, the execution time in milliseconds of the whole BIKE KEM and the breakdown showing the execution times of the three KEM primitives.

Figure 6.9 details the speedup of the reference and proposed HW/SW solutions over software execution, i.e., the ratio between the software execution time and the execution time of each HW/SW instance. Compared to software AES-128- and AES-192-equivalent security BIKE instances, our solutions provide speedups of up to 3.34× and 2.88×, respectively.

Our *Zynq-7010* solution provides a 1.27× speedup over the reference HW/SW one targeting the same chip, while our *Zynq-7015* one results in a 1.98× speedup over the corresponding state-of-the-art reference and a 1.18× speedup over the *Zynq-7020* one. In addition, the performance of the

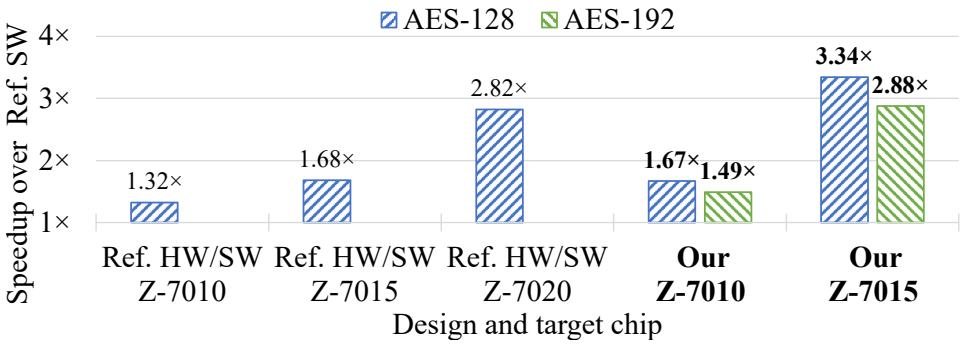


Figure 6.9: *Speedup over the reference software execution.*

proposed *Zynq-7010* instance, which implements the decoding operation in hardware, is almost equivalent to the reference *Zynq-7015* one, which instantiates the whole decapsulation primitive, demonstrating the effectiveness of instantiating in hardware solely the most time-consuming operations and dedicating all the FPGA resources to improving their performance.

6.5 FARMER evaluation

To evaluate the effectiveness of the ML-based exploration methodology discussed in Section 5.2, we conducted an extensive experimental campaign composed of both exhaustive explorations, where the results of all the configurations belonging to the design space were known, and realistic scenarios, where the design space is too large to test every single SoC and the optimal configuration cannot be found. All our execution results have been collected on board.

The rest of this section is divided as follows. Section 6.5.1 describes the hardware, software, and architectural setup employed for our experiments, as well as some details on the computation of the design space size. Section 6.5.2 reports some simulations of our optimization process, carried out using exhaustive data previously collected on an FPGA. Finally, Section 6.5.3 shows a set of optimizations performed with our methodology on larger and more realistic designs.

6.5.1 Experimental setup

Hardware setup The SoCs implemented during our experiments were tested on a Siemens proFPGA quad motherboard (*mb-4m-r3*) [97] equipped with an AMD Virtex-7 2000 (*xc7v2000t-2flg1925-c*) FPGA daughter board (*fm-xctv2000t-r2*) [96]. Virtex-7 2000 is the largest chip of AMD's Virtex-7

Table 6.12: *FPGA resource utilization and throughput of the accelerator tiles. Each accelerator has been instantiated with three different replication factors for the computational core: 1×, 2×, and 4×. Legend:* ***Accel.*** *accelerator,* ***Thr.*** *throughput (MB/s),* ***BR*** *BRAM blocks (18Kb each),* ***CB*** *compute-bound,* ***MB*** *memory-bound.*

Accel.	Type	K	LUT	FF	BR	DSP	Thr.
adpcm	CB	×1	10899	11720	25	81	1.39
		×2	16455	15158	48	162	2.74
		×4	27313	21780	94	324	5.37
aes	CB	×1	7969	10224	28	9	2.09
		×2	10303	11878	54	18	4.08
		×4	14968	15186	106	36	7.85
dfadd	MB	×1	11268	11199	2	9	6.27
		×2	16988	14090	2	18	12.52
		×4	28599	19614	2	36	21.92
gsm	CB	×1	9900	11418	18	62	4.55
		×2	14304	14520	34	124	8.81
		×4	22927	20473	66	248	16.54
sha3	MB	×1	10509	10620	2	1	179.7
		×2	16688	13955	2	2	213.8
		×4	28726	20363	2	4	237.9

family, featuring 1221600 LUTs, 2443200 FFs, 2584 18Kb BRAMs, 2160 DSPs elements, and 24 MMCMs.

Software setup We used the set of accelerators listed in Table 6.12 to validate the proposed methodology. In particular, three of them (*adpcm*, *dfadd*, *gsm*) belong to the CHStone benchmark suite [40], while the remaining ones (*aes*, *sha3*) are handwritten HDL accelerators [48, 50]. The hardware flow inside our experimental campaign was split between SoC implementation on AMD Vivado 2019.2, and FPGA programming on Siemens proFPGA Builder 2019A-SP2. The HLS of the CHStone benchmark applications was executed using AMD Vivado HLS 2019.2. The configuration of the SoC and the training of the model were carried out on a host computer equipped with an Intel i5 processor (*Intel Core i5-12400*) and 64GB of DDR4 RAM, running Kubuntu 22.04.

Use-case architecture The experiments described in this section make use of heterogeneous SoCs based on a 2D mesh NoC topology, although the proposed methodology is independent from the chosen architecture. Thanks

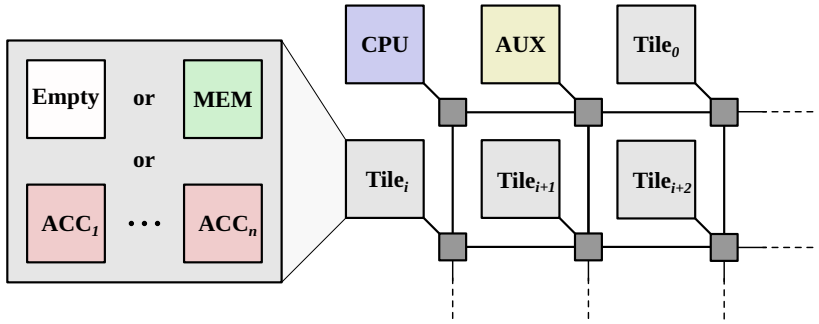


Figure 6.10: *Use-case SoC architecture.*

to the use of a parametric design, the tool can automatically generate a SoC by simply defining the mesh size and the content of each tile.

The specific architecture employed for this work is depicted in Figure 6.10. Each SoC must reserve one tile for a RISC-V, 64-bit CVA6 CPU [112] that runs the main program, and another tile for auxiliary logic that manages I/O interfaces and other SoC-wide functionalities. The other tiles, instead, can implement either an accelerator or a memory channel, and are the target of our optimization process. They can also be left empty to save area. Each accelerator tile executes an entire application and can be parallelized at design time to tune its area-performance trade-off. In our experiments, we let the optimization model choose between three parallelism levels: $r_j \in \{1, 2, 4\}$. On the other side, memory channels are essential to dispose of read and write operations, meaning that the maximum throughput achievable by a particular set of accelerators heavily depends on the number of available memory connections.

The collection of the execution statistics is divided into three phases. During the first one, the bitstream containing the SoC implementation is loaded onto the board, instantiating the design on the FPGA. After the SoC is ready, the second step occurs: a binary containing the testing program is sent to the main memory, and the system is booted. In the third phase, the program running on the CPU starts the execution of all the accelerators, lets them run for several seconds, collects the execution data, and sends them to the host computer through a USB-to-serial interface.

Design space size The size of a design space is computed as follows. The number of free tiles in a SoC is:

$$T = NoC_x \times NoC_y - 2, \tag{6.1}$$

Table 6.13: *Experimental scenarios, characterized by different DSE parameters and SLAs. The double line divides the exhaustive explorations (above) and the realistic scenarios (below).*

	DSE characteristics		Performance Requirements (MB/s)				
SoC	**NoC**	**Space size**	**adpcm**	**aes**	**dfadd**	**gsm**	**sha**
SoC_1	3×3	157	40	-	-	-	-
SoC_2	3×4	52 543	7.5	11	40	-	-
SoC_3	3×4	178 443	5	11	30	15	-
SoC_4	3×4	352 134	5	7	25	15	200
SoC_5	3×4	52 543	5	7	80	-	-

being NoC_x and NoC_y the number of columns and rows of the 2D mesh, respectively, and considering that two tiles are reserved for the CPU and the auxiliary logic. Since a tile can implement a memory channel, an accelerator with a certain parallelism, or can be left empty, the possible implementations for each free tile are:

$$I = N \times N_R + 2, \tag{6.2}$$

where N is the number of applications and N_R is the number of available parallelism levels for an accelerator.

Each configuration is a combination with repetition of I implementations on T tiles. The total number of configurations is then:

$$C'_{I,T} = \frac{(I+T-1)!}{T!(I-1)!}. \tag{6.3}$$

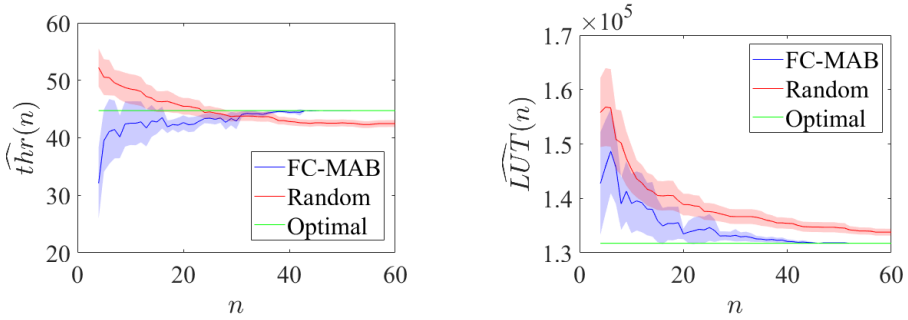
Notably, not all those combinations produce a SoC that satisfy design rules and can execute all the applications. The constraints required for a configuration to be part of the design space are the following:

- there is at least one accelerator for each application;
- there is at least one memory connection;
- the number of memory connections is a power of 2 and is less than or equal to 4.

The first two are functional requirements, while the third is a limitation of the chosen architecture. Thus, the size of the design space of the configurations is the number of combinations $C'_{I,T}$ that satisfy the above-mentioned conditions.

Table 6.14: *Experimental results obtained by running the scenarios listed in Table 6.13. Exhaustive explorations have been run for* 60 *iterations, while realistic scenarios have been run for* 200 *iterations. Notably, for the realistic scenarios the* ***Optimal*** *column does not refer to the absolute optimum (that is unknown), but to the best configuration that the tool can find in the given amount of iterations. The double line divides the exhaustive explorations (above) and the realistic scenarios (below).*

SoC	Strategy	Bootstrap		Results		
		Type	#	First Feasible	Optimal	Explored Space (%)
SoC_1	ML	-	-	3.84 $\pm$ 1.13	14.54 $\pm$ 3.25	9.3
	Random	-	-	2.00 $\pm$ 0.67	51.17 $\pm$ 8.71	32.6
SoC_2	ML	-	-	23	27	5.14×10^{-2}
	ML	Iso	80	21	32	6.09×10^{-2}
	ML	Iso	157	22	52	9.90×10^{-2}
SoC_3	ML	-	-	25	56	3.14×10^{-2}
	ML	Sub3	50	2	2	1.12×10^{-3}
SoC_4	ML	-	-	30	30	8.51×10^{-3}
	ML	Sub3	50	28	60	1.70×10^{-2}
	ML	Sub4	50	24	25	7.10×10^{-3}
SoC_5	ML	-	-	36	36	6.85×10^{-2}



(a) *Throughput.*

(b) *Area.*

Figure 6.11: *Results for the UC1 setting.*

6.5.2 Exhaustive simulation

We provide an example of the working principle of FARMER over a 3×3 NoC running a single application. This allowed us to find the optimal solution using an exhaustive search over data collected on a real board, which gives us the possibility to match our method against the optimal value for the area a^*. We compare the solutions identified over time by the FC-MAB algorithm with the ones provided by an algorithm `Random`, that chooses the configurations randomly over the different rounds. We set the exploration parameter for FC-MAB $b_{j,t-1} = 5$ and we run it over 60 iterations.

We compared the analyzed algorithms in terms of:

- $\widehat{thr}(n)$ [MB/s] average throughput after n rounds of the configuration selected by the algorithm;
- $\widehat{LUT}(n)$ [adimensional] occupied area of the configuration selected by the algorithm,

where the average is taken w.r.t. 50 independent runs of the algorithms. We report the 95% confidence intervals in the plots as semi-transparent areas. We also reported in Table 6.14 the mean ($\pm 95\%$ confidence intervals) of the number of iterations required to find the first feasible solution (**First feasible**), the number of iterations required to find the optimal configuration (**Optimal**), and the percentage of the configuration space we explored on average (**Explored space**).

Results Figure 6.11 provides the performance and area results (in terms of throughput and LUTs, respectively) over the exhaustive scenario. The

`Random` approach provides solutions that fulfill the required SLAs even during the early stages of the exploration process. However, `Random` uses a significantly larger amount of LUTs than those required by the optimal solution, i.e., the red line in Figure 6.11b is always above the optimal green line by a large margin. Finally, even at the end of the exploration process, on average the `Random` strategy does not provide an optimal solution. Such a result was expected since it cannot exploit the information provided by those implementations that have been already selected.

Looking at Table 6.14, we notice that the FC-MAB strategy requires some rounds to find a feasible solution (3.84 iterations on average), but this leads to the identification of the optimal solution in a shorter time (14.54 iterations on average). Thus, the proposed methodology dramatically reduces the time required to find the optimal solution w.r.t. an exhaustive search of the design space C, by minimizing the amount of explored configurations needed to find the optimum of more than 90%. In general, by looking at Figure 6.11, the performance of FC-MAB (blue line) is always closer to the optimum (green) than the `Random` approach (red line). This confirms that our methodology can effectively learn the setting characteristics by exploring just a tiny fraction of the configuration space.

6.5.3 Realistic scenarios

To validate our optimization methodology, we tested it against larger use cases (UC2-UC5), that differ in the number of applications and the SLAs. All the results have been collected for 3×4 SoCs, but the methodology is general and can be applied to any mesh topology. Unlike the exhaustive use cases, we executed each real DSE for a single run (due to the large amount of time required by FPGA implementations), and thus no confidence interval is available.

In such scenarios, we also evaluated the reuse of previously collected exploration data. For this reason, we decided to consider three possible strategies:

1. *base* (no suffix in Table 6.14), when no previous data is available;
2. *isolation* (**ISO** suffix), when the dataset is initialized with some isolation results from 3×3 SoCs. The number of isolation configurations for each application can be 80 or 157 (being 157 the total number of configurations for a 3×3 SoC with one application);
3. *subset* (**SUB** suffix), when the data of a previous exploration with a subset of the application set is available. The figure included in the

suffix indicates the number of applications in the subset. For each run, we randomly selected 50 configurations from smaller UCs and added them to the dataset.

Results From Table 6.14, we observe that in all the attempted runs FARMER identifies a feasible configuration with a few iterations (ranging from 2 to 36 in the realistic scenarios). The same applies to the detection of optimized configurations, that required from 2 to 60 iterations. With respect to an exhaustive search, FARMER finds an optimized configuration exploring just a small fraction of the design space, ranging from the $1.12 \times 10^{-3}\%$ to the $9.90 \times 10^{-2}\%$, saving thousands of computing hours. Notably, no result for the random approach is shown in the lower section of Table 6.14, since the exploration of 200 randomly chosen configurations never returned a feasible solution for any scenario.

Comparing the *base* with the *isolation* and *subset* strategies, we notice that the introduction of previously collected samples decreases the number of iterations required for finding the first feasible solution (e.g., from 30 to 28 and 24 in UC4). However, the *isolation* runs needed a larger number of iterations to find the optimal solution (e.g., from 27 to 32 and 52 in UC2). This suggests that isolation configurations are so different from the ones including all the applications at once that their presence in the dataset may slow the model refinement, since each newly implemented configuration contributes in a smaller part to the entire dataset. Conversely, using the *subset* strategy significantly improves the time required to find the optimal configuration (e.g., from 52 to 2 in UC3). The above-described phenomena can be observed in Figure 6.12, in which we reported the area corresponding to the selected configuration of each method at each iteration. The colored dots represent the first time each optimization process selects a feasible configuration.

6.6 DPR-accelerated DSE

Finally, we conducted a comprehensive experimental campaign combining the ML-driven exploration strategy with the DPR infrastructure, as discussed in Section 5.2.4. We adopted the same approach used in Section 6.5, performing two types of evaluations: *(i)* exhaustive explorations, in which the performance metrics of all configurations were collected in advance and made available to the ML methodology, allowing to perform several independent runs and to know the optimal configuration, and *(ii)* realistic scenarios, characterized by a design space of prohibitive size, where evaluat-

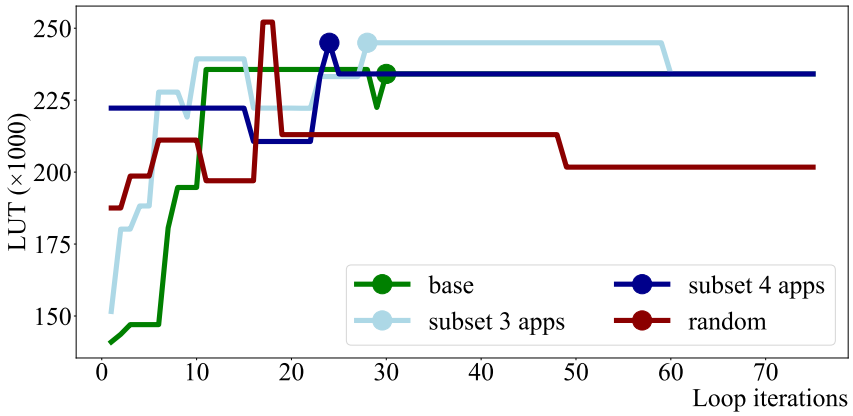


Figure 6.12: *Results of the explorations performed for UC4, with different initialization strategies. The runs were performed for a total of* 200 *iterations, but no change happens after the* 75*th iteration. The dots indicates the first feasible configuration identified by the tool.*

ing every possible SoC configuration is infeasible and the global optimum is not known. All execution results were collected directly on the target hardware platform.

The rest of this section is divided as follows. Section 6.6.1 describes the experimental setup. Section 6.6.2 reports some simulations of our optimization process, carried out using exhaustive data previously collected on the target FPGA. Finally, Section 6.6.3 shows a set of real DSEs performed on larger and more realistic designs.

6.6.1 Experimental setup

Hardware setup The SoCs developed for our experiments were deployed and evaluated on an AMD Alveo U55C PCIe board, which integrates an FPGA from the AMD Virtex UltraScale+ family (*xcu55c-fsvh2892-2L-e*). This FPGA device provides the following hardware resources: 1,303,680 LUTs, 2,607,360 FFs, 4,032 18Kb BRAMs, and 9,024 DSP slices. In addition, it is equipped with 16 GB of on-chip HBM2, which is connected to the FPGA fabric by means of 16 independent memory controllers.

Software setup The set of hardware accelerators used to validate the proposed methodology is summarized in Table 6.15. Specifically, four accelerators–*dfadd*, *dfdiv*, *dfmul*, and *motion*–are derived from the CHStone benchmark suite [40], while the *aes* accelerator is a high-performance cryptographic module implemented in SystemVerilog [48].

Table 6.15: *FPGA resource utilization and throughput of the accelerator tiles. Each accelerator has been instantiated with three different replication factors for the computational core: 1×, 2×, and 4×. Legend:* ***Accel.*** *accelerator,* ***Thr.*** *throughput (MB/s),* ***BR*** *18Kb BRAM blocks,* ***CB*** *compute-bound,* ***MB*** *memory-bound.*

Accel.	Type	K	LUT	FF	BR	DSP	Thr.
aes	MB	×1	9164	11811	0	1	13.28
		×2	12426	15068	0	2	22.54
		×4	18566	21316	0	4	35.71
dfadd	MB	×1	11003	10737	0	9	8.04
		×2	16246	13172	0	18	12.21
		×4	26846	17789	0	36	19.32
dfdiv	CB	×1	10533	11983	0	33	3.29
		×2	14803	15404	0	66	6.22
		×4	23976	22250	0	132	11.73
dfmul	MB	×1	8837	10177	0	25	7.77
		×2	11698	12051	0	50	13.91
		×4	17763	15549	0	100	19.37
motion	CB	×1	9399	10340	10	9	0.67
		×2	12596	12084	20	18	1.11
		×4	19172	15573	40	36	1.68

The hardware implementation flow adopted in this experimental campaign employed AMD Vivado 2023.2 for SoC synthesis, place-and-route, and FPGA programming. The CHStone applications were synthesized using AMD Vivado HLS 2019.2. All SoC implementation procedures and machine learning model training were performed on a host machine equipped with an Intel Core i5-12400 processor and 64 GB of DDR4 RAM, running Ubuntu 22.04.

Use-case architecture The experiments presented in this section target the same 2D-mesh, heterogeneous SoCs described in Section 6.5. The only significant distinction lies in the memory subsystem, which in this case leverages the HBM2 banks integrated into the AMD Alveo U55C board. Referring to Figure 6.10, each memory tile implements an interface with a distinct HBM bank, ensuring that memory transactions are fully parallel and non-interfering across different memory channels.

Table 6.16: *Experimental scenarios, characterized by different DSE parameters and SLAs. The double line divides the exhaustive explorations (above) from the realistic scenarios (below).*

	Size		Performance Requirements (MB/s)				
SoC	**NoC**	**Space**	**aes**	**dfadd**	**dfdiv**	**dfmul**	**motion**
SoC_1	3×3	3294	22	13	–	–	1.5
SoC_2	4×4	727K	45	30	–	–	2.5
SoC_3	4×4	1.96M	38	22	8	–	1.8
SoC_4	4×4	7.50M	30	15	6	15	1.2

6.6.2 Exhaustive scenario

We illustrate the operating principle of the FARMER methodology using a 3×3 NoC executing three concurrent applications. This experimental setup leads to a design space composed of 3294 configurations, which is small enough to be explored exhaustively and thus enabling the identification of the optimal configuration. This allows for a direct comparison between the results obtained by our methodology and the absolute optimal solution, whose area is denoted as a^*.

We evaluate the configurations selected over time by the FC-MAB algorithm employing three distinct machine learning models: Gaussian Process [107] (`GP`), Random Forest [91] (`RF`), and K-Nearest Neighbors [84] (`kNN`). Gaussian Process is a probabilistic model that predicts using a distribution over functions, ideal for small datasets and smooth trends. Random Forest is an ensemble of decision trees that averages results for robust, high-accuracy predictions, handling non-linear data well. K-Nearest Neighbors is an instance-based method that predicts based on the average of the closest data points, making it sensitive to local structure and data scaling. They were implemented with the *scikit-learn* Python library, using the default parameters. As a baseline, we include a purely random selection strategy (`RAND`) that selects configurations uniformly at random across rounds. The exploration parameter in the FC-MAB algorithm is set to $b_{j,t-1} = 5$ in order to allow the model to consider a broad range of configurations, ensuring comprehensive coverage up to the 5-sigma range and reducing the risk of missing potentially optimal configurations. Each algorithm is executed for 50 iterations.

The performance of the algorithms is compared using the following metrics:

Table 6.17: *Experimental results for the explorations targeting the scenarios listed in Table 6.16. Exhaustive explorations have been run for* 50 *iterations, while realistic scenarios have been run for* 200 *iterations. Notably, the* ***Optimized*** *column does not refer to the absolute optimum, but to the best configuration that the tool could find in the given number of iterations. The double line divides the exhaustive explorations (above) and the realistic scenarios (below).* ***UC****: use case.*

		Results			
SoC	**Model**	**First Feasible**	**Optimized**	**Expl. Space (%)**	**Area (#LUTs)**
SoC_1	GP	28.7 $\pm$ 0.66	28.7 $\pm$ 0.66	$8.7 * 10^{-1}\%$	144922 (+0.42%)
	RF	7.60 $\pm$ 2.52	13.3 $\pm$ 2.39	$2.3 * 10^{-1}\%$	145050 (+0.51%)
	kNN	13.3 $\pm$ 3.16	28.5 $\pm$ 3.37	$4.0 * 10^{-1}\%$	145102 (+0.55%)
SoC_2	GP	87 (102m)	87 (102m)	$1.2 * 10^{-2}\%$	246124
	RF	53 (51m)	115 (111m)	$7.3 * 10^{-3}\%$	246124
	kNN	23 (22m)	141 (134m)	$3.2 * 10^{-3}\%$	255999
SoC_3	GP	39 (56m)	66 (95m)	$2.0 * 10^{-3}\%$	224663
	RF	4 (5.4m)	26 (35m)	$2.0 * 10^{-4}\%$	223227
	kNN	39 (52m)	128 (171m)	$2.0 * 10^{-3}\%$	224565
SoC_4	GP	29 (79m)	29 (79m)	$3.9 * 10^{-4}\%$	211003
	RF	34 (86m)	41 (104m)	$4.5 * 10^{-4}\%$	221836
	kNN	14 (35m)	56 (141m)	$1.9 * 10^{-4}\%$	222175

- $\widehat{thr}(n)$ [MB/s]: the average throughput after n rounds, evaluated for the configuration selected by the algorithm;

- $\widehat{LUT}(n)$ [adimensional]: the area occupied by the selected configuration, measured in LUTs.

Averaging is performed over 30 independent runs for each algorithm. The 95% confidence intervals are depicted in the figures as semi-transparent areas.

Table 6.16 reports the results, including the mean number of iterations (with 95% confidence intervals) required to identify the first feasible configuration (**First feasible**) and the best one proposed by the model (**Optimized**), as well as the fraction of the design space that has been explored to find the first feasible configuration (**Expl. space**) and the area of the best configuration with the relative LUT difference compared to the global optimum (**Area**).

Results Figure 6.13 presents the performance and area results (in terms of throughput and LUT usage) obtained under the exhaustive search scenario. The `RAND` strategy fails to reliably identify feasible solutions, achieving success in only 14 out of 30 runs. This highlights the ineffectiveness of unguided exploration even in relatively small design spaces consisting of a few thousand configurations. Furthermore, as shown in Figure 6.13a, `RAND` consistently selects configurations with significantly higher LUT usage compared to the optimal solution, as indicated by the red curve remaining well above the optimal green reference line.

In contrast, the FC-MAB algorithm successfully identifies feasible configurations in all runs and with all three ML models. As shown in Table 6.16, the number of iterations required to reach feasibility varies depending on the model. The `RF` model demonstrates the fastest convergence, requiring on average only 7.6 iterations to find a feasible solution and 13.3 iterations to reach a preferred configuration. All three models are able to identify configurations with area usage very close to the optimal, with a relative deviation of $\sim 0.5\%$.

The exploration dynamics of the models, as shown in Figure 6.13, reveal distinct strategies. The `GP` model adopts a conservative search behavior, beginning with smaller and lower-performance configurations and gradually increasing throughput and area. In many cases, the first feasible configuration is also the best one that the model is able to identify. In contrast, the `RF` and `kNN` models tend to overshoot, initially selecting high-performance and larger configurations and then iteratively refining toward lower-area solutions. This behavior enables them to discover feasible configurations faster than the `GP` model.

Table 6.18: *Time breakdown (in seconds) for the GP-based explorations in the UC2, UC3, and UC4 scenarios.*

	Average time per iteration			
Scenario	**Prediction**	**Reconfiguration**	**Execution**	**Total**
UC2	28 (41%)	27 (40%)	13 (19%)	68
UC3	47 (54%)	27 (31%)	13 (15%)	87
UC4	127 (76%)	28 (17%)	13 (8%)	168

6.6.3 Realistic scenarios

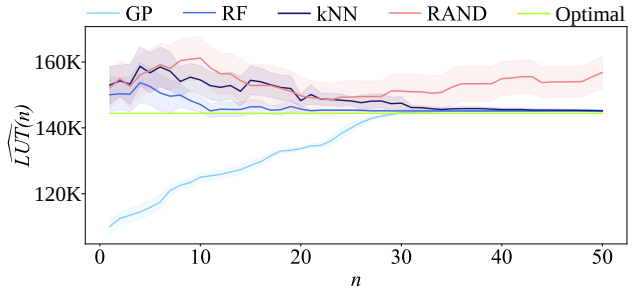
To further validate the proposed methodology, we extended our evaluation to larger use cases (UC2-UC4), targeting 4×4 SoC configurations. Due to the large time, in the order of a few hours, required to perform real DSEs (which, unlike the exhaustive scenario, cannot rely on pre-collected performance data), each exploration was executed only once. Consequently, confidence intervals are not reported. Given the huge size of the design space, the global optimum remains unknown, as it is not possible to perform an exhaustive exploration in a reasonable time. The outcomes of the realistic DSEs are presented in Table 6.16. In this case, we also report the time, expressed in minutes, required by the framework to identify the first feasible and the optimized configuration.

Results As shown in Table 6.17, in all the evaluated runs FARMER was able to identify a feasible configuration within a limited number of iterations, ranging from 4 to 87. Similarly, optimized configurations were found within 26 to 141 iterations, depending on the specific use case. The portion of the design space explored during these DSEs ranged from $1.9 \times 10^{-4}\%$ to $1.2 \times 10^{-2}\%$, representing a significant reduction in exploration time compared to an exhaustive search, allowing to save thousands of runtime hours. Notably, the results for the random selection strategy are not reported in Table 6.17, as the random choice of 200 configurations never yielded a feasible solution across any scenario.

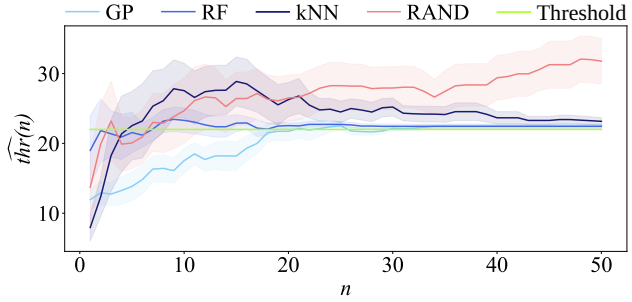
Table 6.17 also details the time required to identify the first feasible configuration. Compared with the results reported in 6.5, it is evident that the integration of the DPR technique reduces the deployment time by more than an order of magnitude. This substantial decrease in the configuration deployment time shifts the performance bottleneck toward the prediction phase of the ML models. Since the regressors evaluate the performance of each configuration in the design space, the computational workload of

prediction becomes increasingly dominant as the design space scales to millions of configurations. This is exemplified by Table 6.18, which reports the average time per iteration in the GP-based realistic scenarios. While the reconfiguration and execution times are constant, the prediction time strongly depends on the design space size.

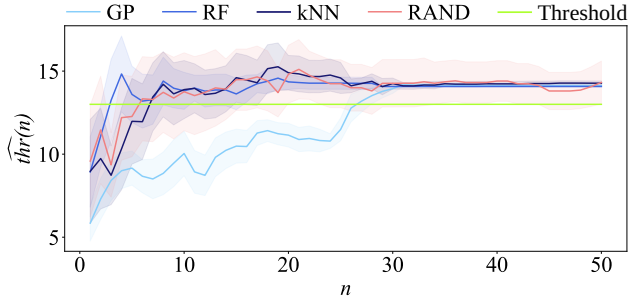
Among the three tested models, the `GP` demonstrated superior consistency in identifying high-quality configurations. This is particularly notable in UC4, where both the `RF` and `kNN` models converged on suboptimal configurations, exceeding the LUT usage of the `GP` solution by over 10,000 LUTs. Nevertheless, the more stochastic behavior of `RF` and `kNN` enables faster initial exploration, allowing them to locate feasible configurations earlier than the more conservative `GP`. Overall, the results obtained in these larger scenarios are consistent with the findings from the exhaustive evaluations. Moreover, they confirm that the proposed methodology scales effectively to complex SoC architectures, even when the design space contains millions of configurations. Figure 6.14 illustrates the DSE results for the UC2 scenario, confirming the behavioral patterns of the three ML models observed in Figure 6.13.



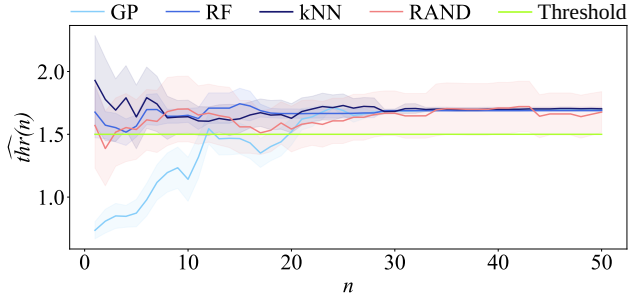
(a) *Area.*



(b) *aes throughput.*



(c) *dfadd throughput.*



(d) *motion throughput.*

Figure 6.13: *Area and throughput of the best identified configuration at each iteration of the optimization flow for the exhaustive scenario UC1.*

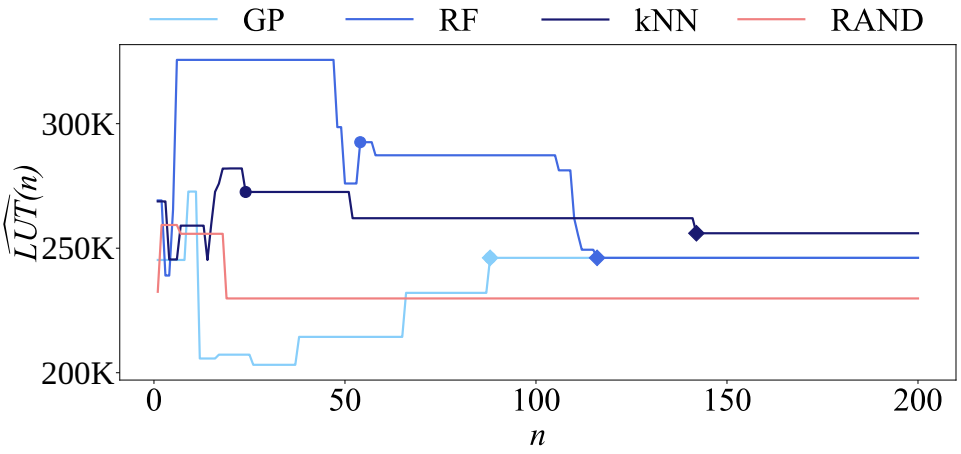


Figure 6.14: *Area consumption (in LUTs) for the best configuration at each iteration for the DSEs targeting UC2. The round and diamond markers denote the first feasible and the preferred configuration, respectively.*

CHAPTER 7

Conclusions

Data centers are the backbone of today's digital world, but the limits of CPUs and the energy demands of GPUs call for new solutions. FPGAs offer a unique balance of performance, flexibility, and efficiency, making them an increasingly attractive option for hyperscale computing. Their rise has been fueled by advances in logic capacity, memory integration, and interconnects, as well as successful deployments by major cloud providers.

Yet, the road to widespread adoption is hindered by design complexity, underutilization, and security risks. Research has responded with approaches like multi-tenancy, SoC prototyping, and specialized FPGA-based accelerators. Still, the vast design space of heterogeneous architectures remains a key challenge.

The goal of this thesis is to provide new solutions for this optimization problem. We adopted a two-fold approach, leading in two orthogonal directions.

On one side, we propose the first DSE-oriented SoC prototyping platform, that is a framework that facilitates the automatic generation and testing of heterogeneous SoCs, with features that enhance the tuning of the SoC PPA profile as well as the observability of runtime statistics. This is allowed by a set of features that are not available in other academic SoC prototyping

frameworks, namely the DFS infrastructure, the multi-replica architecture, the memory benchmarking platform, and the DPR mechanism.

On the other side, we identified methodologies that help the finding of the optimal configuration on a SoC, both at intra- and inter-application level. The former is approached by using an HW/SW co-design strategy to identify which operations should be executed in software and which should be accelerated in hardware, given the resource limitations of the device. The latter is instead elaborated as a placement problem, which we have solved using an online-learning-based methodology that can automatically detect the best accelerators allocation to satisfy a specific optimization problem.

The combination of these two approaches generates a complete HW/SW flow that can exploit the speed and automation of the DPR-powered SoC prototyping platform with the efficiency of an ML strategy that needs just a minimal fraction of the configurations to identify the best one. Moreover, this thesis presents a large set of experimental results that prove the efficacy of this methodology over a large variety of hardware architectures and experimental scenarios.

The broad scope of this research leaves room for several meaningful extensions. A straightforward direction would be to enhance the proposed platform with support for multi-FPGA systems. This would increase the available computational resources and enable the execution of larger and more complex applications. At the same time, such an extension would introduce new challenges. The design space would grow significantly, and communication between devices could become a critical bottleneck that must be carefully modeled and optimized.

Another possible extension is to broaden the framework to include additional computing architectures, such as CPUs and GPUs. Incorporating heterogeneous resources would make the platform more representative of modern data center environments, where applications are rarely executed on a single type of hardware. This would also allow the methodology to explore trade-offs across different architectural choices.

Finally, future work could incorporate power consumption as an explicit optimization metric. Power constraints are increasingly relevant in data centers, where electricity and cooling costs have a substantial economic and environmental impact. Including power-aware optimization would therefore improve the practical relevance of the proposed approach.

Appendices

APPENDIX $\mathcal{A}$

List of publications

This appendix presents all the articles published during the PhD, which together constitute the core of the thesis. Each publication is directly connected to the research carried out during this doctoral work, and for most of them the author of this thesis conducted the majority of design, testing, and experimental activities, while also contributing to the writing of all the papers. For each article, publication details are provided (journal or conference, year, volume, and page numbers), along with the digital object identifier (DOI) and a description of the author's specific contributions.

- **Title:** Hardware-software co-design of bike with hls-generated accelerators.
  Authors: Gabriele Montanaro, Andrea Galimberti, Ernesto Colizzi, and Davide Zoni.
  Book title: Proceedings of the 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS).
  Year: 2022
  Pages: 1-4
  DOI: 10.1109/ICECS202256217.2022.9970992.
  Cited as: [71].

Personal contribution: Contributed to the design, implementation, and testing. Also contributed to the writing of the paper.

- **Title:** HLS-based acceleration of the BIKE post-quantum KEM on embedded-class heterogeneous SoCs.
  Authors: Andrea Galimberti, Gabriele Montanaro, and Davide Zoni.
  Book title: Proceedings of the 30th IEEE International Conference on Electronics, Circuits and Systems (ICECS).
  Year: 2023
  Pages: 1-4
  DOI: 10.1109/ICECS58634.2023.10382918.
  Cited as: [36].
  Personal contribution: Contributed to the design, implementation, and testing. Also contributed to the writing of the paper.

- **Title:** A Prototype-Based Framework to Design Scalable Heterogeneous SoCs with Fine-Grained DFS.
  Authors: Gabriele Montanaro, Andrea Galimberti, and Davide Zoni.
  Book title: Proceedings of the 42nd IEEE International Conference on Computer Design (ICCD).
  Year: 2024
  Pages: 681-684
  DOI: 10.1109/ICCD63220.2024.00108.
  Cited as: [72].
  Personal contribution: Contributed to the design, implementation, and testing. Also contributed to the writing of the paper.

- **Title:** FARMER: An Online-Learning Driven Methodology for Workload Consolidation on Large FPGAs.
  Authors: Gabriele Montanaro, Francesco Trovò, and Davide Zoni.
  Book title: Proceedings of the 57th IEEE International Symposium on Circuits and Systems (ISCAS).
  Year: 2025
  Pages: 1-5
  DOI: 10.1109/ISCAS56072.2025.11043238.
  Cited as: [74].
  Personal contribution: Contributed to the design, implementation, and testing. Also contributed to the writing of the paper.

- **Title:** Benchmarking Platform for DDR4 Memory Performance in Data-Center-Class FPGAs.

Authors: Andrea Galimberti, Gabriele Montanaro, Andrea Motta, Federico Proverbio, and Davide Zoni.
Book title: Proceedings of the 57th IEEE International Symposium on Circuits and Systems (ISCAS).
Year: 2025
Pages: 1-5
DOI: 10.1109/ISCAS56072.2025.11043686.
Cited as: [35].
Personal contribution: Supervised the design and implementation of the prototype. Also contributed to the writing of the paper.

- **Title:** FARMER: Online-Learning-Based Workload Consolidation on Large FPGAs Accelerated With Dynamic Partial Reconfiguration.
  Authors: Gabriele Montanaro, Francesco Trovò, and Davide Zoni.
  Journal: IEEE Transactions on Very Large Scale Integration (VLSI) Systems.
  Year: 2025
  Pages: 1-9
  DOI: 10.1109/TVLSI.2025.3594952.
  Cited as: [75].
  Personal contribution: Contributed to the design, implementation, and testing. Also contributed to the writing of the paper.

- **Title:** Omega: A Hardware-Software Framework for Complete Design Space Exploration of FPGA-Based Heterogeneous Multi-Core SoCs.
  Authors: Gabriele Montanaro, Andrea Galimberti, and Davide Zoni.
  Journal: IEEE Transactions on Computers.
  Year: 2025
  Pages: 1-13
  DOI: 10.1109/TC.2025.3624704.
  Cited as: [73].
  Personal contribution: Contributed to the design, implementation, and testing. Also contributed to the writing of the paper.

List of figures

1.1 FPGA size comparison. 3
1.2 FPGA adoption in data centers timeline. 4
1.3 Design space exploration example. 5
1.4 Effect of traffic on accelerator throughput. 7
1.5 An overview of the proposed DSE flow. 10

2.1 A 3×3 NoC. 14
2.2 Traditional DSE flow on FPGAs. 18

3.1 Example SoC generated with ESP. 26

4.1 Architecture of a generic Omega SoC. 36
4.2 DFS actuator architecture. 38
4.3 Multi-replica accelerator-tile architecture. 40
4.4 Overview of Omega's DPR architecture 42
4.5 Architecture of the memory benchmarking platform. 44
4.6 Flow of the Omega DSE framework 48

5.1 Overview of FARMER. 55
5.2 The FARMER online learning pipeline. 59

6.1 Architecture and floorplan of a 4×4 SoC. 67
6.2 Accelerators characterization. 70
6.3 Throughput and round-trip time of two accelerators. 71
6.4 Analysis of memory traffic. 74

6.5 Throughput of single-channel memory configurations. . . . 78
6.6 Breakdown of DDR4 throughput. 79
6.7 Experimental SoC architecture for exhaustive exploration. . 80
6.8 Heterogeneous SoC architecture for HW/SW co-design. . . 88
6.9 BIKE execution speedup over the software implementation. 92
6.10 Use-case SoC architecture for FARMER evaluation. 94
6.11 DSE results for the UC1 setting. 97
6.12 DSE results for UC4 scenario. 100
6.13 Area and throughput optimization for the exhaustive use case. 107
6.14 DSE results for UC2 scenario. 108

List of tables

3.1 SoC platforms comparison. 24
3.2 SoC optimization methodologies comparison. 29
3.3 DFS and DVFS infrastructures comparison. 31
3.4 Memory benchmarking platforms comparison. 33

4.1 Configurability of the memory benchmarking platform. . . . 45

6.1 Accelerators list. 64
6.2 Accelerators for Omega characterization. 65
6.3 Frequency islands resource utilization. 68
6.4 Resource utilization of the memory benchmarking platform. 76
6.5 Single-channel configuration DDR4 throughput. 77
6.6 Accelerators for the DPR experimental evaluation. 81
6.7 Scenarios for the DPR experimental evaluation. 82
6.8 Exhaustive exploration time. 84
6.9 BIKE's software execution times breakdown. 87
6.10 Breakdown of BIKE's FPGA resource utilization. 90
6.11 BIKE's hardware execution times breakdown. 91
6.12 Accelerators for FARMER's experimental campaign. 93
6.13 Experimental scenarios for FARMER's explorations. 95
6.14 FARMER's experimental results. 96
6.15 Accelerators for the evaluation of the full DSE flow. 101
6.16 Full DSE flow scenarios. 102
6.17 Full DSE flow results. 103

6.18 Exploration time breakdown. 105

Bibliography

[1] 7 series product tables and product selection guide. https://docs.amd.com/v/u/en-US/7-series-product-selection-guide. Accessed: 2025-03-10.

[2] Baidu cloud compute (bcc). https://intl.cloud.baidu.com/en/product/bcc.html. Accessed: 2025-11-03.

[3] Data center: market data and analysis. https://www.statista.com/study/107178/data-center/. Accessed: 2025-03-10.

[4] Deep dive into alibaba cloud f3 fpga as a service instances. https://www.alibabacloud.com/blog/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances_594057. Accessed: 2025-11-03.

[5] Fp1-based fpga development suite. https://github.com/huaweicloud/huaweicloud-fpga. Accessed: 2025-11-03.

[6] Fpga development with amazon ec2 f1. https://fpga-development-on-ec2.workshop.aws/en/. Accessed: 2025-03-10.

[7] Versal premium series product selection guide. https://docs.amd.com/v/u/en-US/versal-premium-psg. Accessed: 2025-03-10.

[8] What is tencent cloud fpga cloud server? https://www.tencentcloud.com/techpedia/101401. Accessed: 2025-11-03.

[9] Np sizes series. Microsoft Learn, 2024. Accessed: 2025-11-03.

[10] Advanced Micro Devices, Inc. UltraScale FPGA Product Tables and Product Selection Guide (XMP102), 2016.

[11] Advanced Micro Devices, Inc. UltraScale Architecture-Based FPGAs Memory IP Product Guide (PG150), 2022.

[12] Lamees M. Al Qassem, Thanos Stouraitis, Ernesto Damiani, and Ibrahim M. Elfadel. Fpgaaas: A survey of infrastructures and systems. *IEEE Transactions on Services Computing*, 15(2):1143–1156, 2022.

[13] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanoviç, and Borivoje Nikoliç. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21, 2020.

[14] Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Valentin Vasseur, and Gilles Zémor. BIKE website. `https://www.bikesuite.org/`, 2021.

[15] Arm Limited. AMBA AXI and ACE Protocol Specification Version H.c, 2021.

[16] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, Apr 2016.

[17] Integrated Systems Laboratory at ETH Zürich and Energy efficient Embedded Systems at University of Bologna. Pulp website. `https://pulp-platform.org`, 2013. [Online; accessed 11-October-2025].

[18] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47:235–256, 2002.

[19] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th annual design automation conference*, pages 1216–1225, 2012.

[20] Jeff Barr. Aqua (advanced query accelerator) – a speed boost for your amazon redshift queries. AWS News Blog, 2021. Accessed: 2025-11-03.

[21] Jeff Barr. Now available – second-generation fpga-powered amazon ec2 instances (f2). AWS News Blog, 2024. Accessed: 2025-11-03.

[22] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011.

[23] Christophe Bobda, Joel Mandebi Mbongue, Paul Chow, Mohammad Ewais, Naif Tarafdar, Juan Camilo Vega, Ken Eguro, Dirk Koch, Suranga Handagala, Miriam Leeser, Martin Herbordt, Hafsah Shahzad, Peter Hofste, Burkhard Ringlein, Jakub Szefer, Ahmed Sanaullah, and Russell Tessier. The future of fpga acceleration in datacenters and the cloud. *ACM Trans. Reconfigurable Technol. Syst.*, 15(3), feb 2022.

[24] Paul Bogdan, Radu Marculescu, Siddharth Jain, and Rafael Tornero Gavila. An optimal control approach to power management for multi-voltage and frequency islands multiprocessor platforms under highly variable workloads. In *2012 IEEE/ACM Sixth international symposium on networks-on-chip*, pages 35–42. IEEE, 2012.

[25] Marcelo Brandalero, Thiago Dadalt Souto, Luigi Carro, and Antonio Carlos Schneider Beck. Predicting performance in multi-core systems with shared reconfigurable accelerators. *Journal of Systems Architecture*, 98:201–213, 2019.

[26] Iulian Brumar, Georgios Zacharopoulos, Yuan Yao, Saketh Rama, David Brooks, and Gu-Yeon Wei. Early dse and automatic generation of coarse-grained merged accelerators. *ACM Transactions on Embedded Computing Systems*, 22(2):1–29, 2023.

[27] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM international symposium on microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[28] Yen-Lung Chen, Bo-Yi Chang, Chia-Hsiang Yang, and Tzi-Dar Chiueh. A high-throughput fpga accelerator for short-read mapping of the whole human genome. *IEEE Transactions on Parallel and Distributed Systems*, 32(6):1465–1478, 2021.

[29] Thanh Cong and François Charot. Design space exploration of heterogeneous-accelerator socs with hyperparameter optimization. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pages 338–343, 2021.

[30] Francesco Conti, Davide Rossi, Antonio Pullini, Igor Loi, and Luca Benini. Pulp: A ultra-low power parallel accelerator for energy-efficient and flexible embedded vision. *Journal of Signal Processing Systems*, 84(3):339–354, 2016.

[31] Bouthaina Damak, Rachid Benmansour, Mouna Baklouti, Smail Niar, and Mohamed Abid. Design space exploration for customized asymmetric heterogeneous mpsoc. In *2014 17th Euromicro Conference on Digital System Design*, pages 50–57. IEEE, 2014.

[32] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, et al. The celerity open-source 511-core risc-v tiered accelerator fabric: Fast architectures and design methodologies for fast chips. *IEEE Micro*, 38(2):30–41, 2018.

[33] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. Leblanc. Design of ion-implanted mosfet's with very small physical dimensions. *Proceedings of the IEEE*, 87(4):668–678, 1999.

[34] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2018.

[35] Andrea Galimberti, Gabriele Montanaro, Andrea Motta, Federico Proverbio, and Davide Zoni. A benchmarking platform for ddr4 memory performance in data-center-class fpgas. In *2025 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2025.

[36] Andrea Galimberti, Gabriele Montanaro, and Davide Zoni. Hls-based acceleration of the bike post-quantum kem on embedded-class heterogeneous socs. In *2023 30th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 1–4, 2023.

[37] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017.

[38] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.

[39] Davide Giri, Kuan-Lin Chiu, Guy Eichler, Paolo Mantovani, and Luca P. Carloni. Accelerator integration for open-source soc design. *IEEE Micro*, 41(4):8–14, 2021.

[40] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1192–1195, 2008.

[41] Sebastian Herbert, Siddharth Garg, and Diana Marculescu. Exploiting process variability in voltage/frequency control. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(8):1392–1404, 2011.

[42] Robert Hesse and Natalie Enright Jerger. Improving dvfs in nocs with coherence prediction. In *Proceedings of the 9th International Symposium on Networks-on-Chip*, pages 1–8, 2015.

[43] Hongjing Huang, Zeke Wang, Jie Zhang, Zhenhao He, Chao Wu, Jun Xiao, and Gustavo Alonso. Shuhai: A Tool for Benchmarking High Bandwidth Memory on FPGAs. *IEEE Transactions on Computers*, 71(5):1133–1144, 2022.

[44] Bruce Jacob, David Wang, and Spencer Ng. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2010.

[45] JEDEC. JEDEC: Global Standards for the Microelectronics Industry. `https://www.jedec.org/`, 2025.

[46] Zai Jian Jia, Andy D Pimentel, Mark Thompson, Tomás Bautista, and Antonio Núñez. Nasa: A generic infrastructure for system-level mp-soc design space exploration. In *2010 8th IEEE Workshop on Embedded Systems for Real-Time Multimedia*, pages 41–50. IEEE, 2010.

[47] Weixiong Jiang, Heng Yu, and Yajun Ha. Enabling fine-grained dynamic voltage and frequency scaling in sdsoc. In *2019 32nd IEEE International System-on-Chip Conference (SOCC)*, pages 56–61. IEEE, 2019.

[48] Joachim Strömbergson. Verilog implementation of the symmetric block cipher AES (NIST FIPS 197). `https://github.com/secworks/aes`, 2014. [Online; accessed 11-January-2024].

[49] Michael Guilherme Jordan, Guilherme Korol, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck. Resource-aware collaborative allocation for cpu-fpga cloud environments. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 68(5):1655–1659, 2021.

[50] Josh Moles. Keccak Verilog Implementation. `https://github.com/jmoles/keccak-verilog/tree/main`, 2013. [Online; accessed 10-May-2024].

[51] Ahmed Kamaleldin and Diana Göhringer. Agiler: An adaptive heterogeneous tile-based many-core architecture for risc-v processors. *IEEE Access*, 10:43895–43913, 2022.

[52] Sukhun Kang and Rakesh Kumar. Magellan: a search and machine learning-based framework for fast multi-core design space exploration and optimization. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1432–1437, 2008.

[53] Yi-Chieh Kao, Hung-An Chen, and Hsi-Pin Ma. An fpga-based high-frequency trading system for 10 gigabit ethernet with a latency of 433 ns. In *2022 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–4. IEEE, 2022.

[54] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42, 2018.

[55] Brucek Khailany, Rangharajan Venkatesan, Jason Clemons, Joel S Emer, Matthew Fojtik, Alicia Klinefelter, Michael Pellauer, Nathaniel Pinckney, Yakun Sophia Shao, Shreesha Srinath, et al. A modular digital vlsi flow for high-productivity soc design. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[56] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with {AmorphOS}. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 107–127, 2018.

[57] Sohyeon Kim, Injun Choi, Minkyu Je, and Ji-Hoon Kim. Dynamic resource management in reconfigurable soc for multi-tenancy support. In *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2024.

[58] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, 2014.

[59] George Kornaros and Dionisios Pnevmatikatos. Dynamic power and thermal management of noc-based heterogeneous mpsocs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(1):1–26, 2014.

[60] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do os abstractions make sense on fpgas? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 991–1010, 2020.

[61] Andreas Kurth, Pirmin Vogel, Alessandro Capotondi, Andrea Marongiu, and Luca Benini. Hero: Heterogeneous embedded research platform for exploring risc-v manycore accelerators on fpga. *arXiv preprint arXiv:1712.06497*, 2017.

[62] Sakari Lahti, Panu Sjövall, Jarno Vanne, and Timo D Hämäläinen. Are we there yet? a study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):898–911, 2018.

[63] Chris Lavin and Alireza Kaviani. Rapidwright: Enabling custom crafted implementations for fpgas. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 133–140. IEEE, 2018.

[64] Harrison Liew, Daniel Grubb, John Wright, Colin Schmidt, Nayiri Krzysztofowicz, Adam Izraelevitz, Edward Wang, Krste Asanović, Jonathan Bachrach, and Borivoje Nikolić. Hammer: A modular and reusable physical design flow tool: Invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, pages 1335–1338, New York, NY, USA, 2022. Association for Computing Machinery.

[65] Gaye Lightbody, Valeriia Haberland, Fiona Browne, Laura Taggart, Huiru Zheng, Eileen Parkes, and Jaine K Blayney. Review of applications of high-throughput sequencing in personalized medicine: barriers and facilitators of future progress in research and clinical application. *Briefings in bioinformatics*, 20(5):1795–1811, 2019.

[66] Alec Lu, Zhenman Fang, Weihua Liu, and Lesley Shannon. Demystifying the memory system of modern datacenter fpgas for software programmers through microbenchmarking. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 105–115, New York, NY, USA, 2021. Association for Computing Machinery.

[67] Paolo Mantovani, Emilio G Cota, Kevin Tien, Christian Pilato, Giuseppe Di Guglielmo, Ken Shepard, and Luca P Carloni. An fpga-based infrastructure for fine-grained dvfs analysis in high-performance embedded systems. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[68] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. Agile soc development with open esp. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, New York, NY, USA, 2020. Association for Computing Machinery.

[69] Joel Mandebi Mbongue, Danielle Tchuinkou Kwadjo, Alex Shuping, and Christophe Bobda. Deploying multi-tenant fpgas within linux-based cloud infrastructure. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(2):1–31, 2021.

[70] Sparsh Mittal and Jeffrey S Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):1–35, 2015.

[71] Gabriele Montanaro, Andrea Galimberti, Ernesto Colizzi, and Davide Zoni. Hardware-software co-design of bike with hls-generated accelerators. In *2022 29th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 1–4, 2022.

[72] Gabriele Montanaro, Andrea Galimberti, and Davide Zoni. A prototype-based framework to design scalable heterogeneous socs with fine-grained dfs, 2024.

[73] Gabriele Montanaro, Andrea Galimberti, and Davide Zoni. Omega: A hardware-software framework for complete design space exploration of fpga-based heterogeneous multi-core socs. *IEEE Transactions on Computers*, pages 1–13, 2025.

[74] Gabriele Montanaro, Francesco Trovò, and Davide Zoni. Farmer: An online-learning driven methodology for workload consolidation on large fpgas. In *2025 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2025.

[75] Gabriele Montanaro, Francesco Trovò, and Davide Zoni. Farmer: Online-learning-based workload consolidation on large fpgas accelerated with dynamic partial reconfiguration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–9, 2025.

[76] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[77] Onur Mutlu and Jeremie S. Kim. RowHammer: A Retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1555–1571, 2020.

[78] National Institute of Standards and Technology (NIST) - U.S. Department of Commerce. Nistir 8413, status report on the third round of the nist post-quantum cryptography standardization process. `https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413.pdf`, 2022.

[79] Ataberk Olgun, Hasan Hassan, A. Giray Yağlıkçı, Yahya Can Tuğrul, Lois Orosa, Haocong Luo, Minesh Patel, Oğuz Ergin, and Onur Mutlu. DRAM Bender: An Extensible and Versatile FPGA-Based Infrastructure to Easily Test State-of-the-Art DRAM Chips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(12):5098–5112, 2023.

[80] Alessandro Ottaviano, Thomas Benz, Paul Scheffler, and Luca Benini. Cheshire: A lightweight, linux-capable risc-v host platform for domain-specific accelerator plug-in. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 70(10):3777–3781, 2023.

[81] Maurizio Palesi and Tony Givargis. Multi-objective design space exploration using genetic algorithms. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 67–72, 2002.

[82] Elias Perdomo, Xavier Martorell, Teresa Cervero, and Behzad Salami. Memory Sandbox: A Versatile Tool for Analyzing and Optimizing HBM Performance in FPGA. In *2024 IEEE 36th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 206–217, 2024.

[83] Matteo Perotti, Samuel Riedel, Matheus Cavalcante, and Luca Benini. Spatz: Clustering compact risc-v-based vector units to maximize computing efficiency. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2025.

[84] Leif E Peterson. K-nearest neighbor. *Scholarpedia*, 4(2):1883, 2009.

[85] Daniel Petrisko, Farzam Gilani, Mark Wyse, Dai Cheol Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, Tavio Guarino, Ajay Joshi, Mark Oskin, and Michael Bedford Taylor. Blackparrot: An agile open-source risc-v multicore for accelerator socs. *IEEE Micro*, 40(4):93–102, 2020.

[86] Alok Prakash, Christopher T Clarke, Siew-Kei Lam, and Thambipillai Srikanthan. Rapid memory-aware selection of hardware accelerators in programmable soc design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(3):445–456, 2017.

[87] Krishna K Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: fine-grained power management for multi-core systems. *ACM SIGARCH Computer Architecture News*, 37(3):302–313, 2009.

[88] Tiago Santos, João Bispo, and João MP Cardoso. A cpu-fpga holistic source-to-source compilation approach for partitioning and optimizing c/c++ applications. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 320–322. IEEE, 2023.

[89] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, 2017.

[90] Pasquale Davide Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, and Luca Benini. Quentin: an ultra-low-power pulpissimo soc in 22nm fdx. In *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, pages 1–3. IEEE, 2018.

[91] Mark R Segal. Machine learning benchmarks and random forest regression. 2004.

[92] Biruk Seyoum, Davide Giri, Kuan-Lin Chiu, Bryce Natter, and Luca Carloni. Pr-esp: An open-source platform for design and programming of partially reconfigurable socs. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2023.

[93] Biruk Seyoum, Marco Pagani, Alessandro Biondi, and Giorgio Buttazzo. Automating the design flow under dynamic partial reconfiguration for hardware-software co-design in fpga soc. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, pages 481–490, New York, NY, USA, 2021. Association for Computing Machinery.

[94] Junnan Shan, Mario R Casu, Jordi Cortadella, Luciano Lavagno, and Mihai T Lazarescu. Exact and heuristic allocation of multi-kernel applications to multi-fpga platforms. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[95] Md Abu Bakar Siddik, Arman Shehabi, and Landon Marston. The environmental footprint of data centers in the united states. *Environmental Research Letters*, 16(6):064017, 2021.

[96] Siemens Electronic Design Automation GmbH. FPGA Module XC7V2000T. `https://www.profpga.com/products/fpga-modules-overview/virtex-7-based/profpga-xc7v2000t`. [Online; accessed 10-May-2024].

[97] Siemens Electronic Design Automation GmbH. proFPGA quad Motherboard for multi-FPGA System. `https://www.profpga.com/products/motherboards-overview/profpga-quad`. [Online; accessed 10-May-2024].

[98] Siemens Electronic Design Automation GmbH. proFPGA DDR4 2.5 GB SDRAM Memory Board, 2022.

[99] Siemens Electronic Design Automation GmbH. proFPGA XCKU115 Kintex UltraScale FPGA Module, 2022.

[100] Niranjan Srinivas, Andreas Krause, Sham Kakade, and Matthias Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. In *Proceedings of the International Conference on Machine Learning*, pages 1015–1022, 2010.

[101] F. Sturesson, J. Gaisler, R. Ginosar, and T. Liran. Radiation characterization of a dual core leon3-ft processor. In *2011 12th European Conference on Radiation and Its Effects on Components and Systems*, pages 938–944, 2011.

[102] Rafael Tornero, David Rodríguez, José M. Martínez, and José Flich. An open-source fpga platform for shared-memory heterogeneous many-core architecture exploration. In *2023 38th Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6, 2023.

[103] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. Heterogeneous resource-elastic scheduling for cpu+ fpga architectures. In *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, pages 1–6, 2019.

[104] Anuj Vaishnav, Khoa Dang Pham, Dirk Koch, and James Garside. Resource elastic virtualization for fpgas using opencl. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 111–1117. IEEE, 2018.

[105] Kizheppatt Vipin and Suhaib A Fahmy. Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Computing Surveys (CSUR)*, 51(4):1–39, 2018.

[106] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. {FpgaNIC}: An {FPGA-based} versatile 100gb {SmartNIC} for {GPUs}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 967–986, 2022.

[107] Christopher KI Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA, 2006.

[108] Yuanlong Xiao, Aditya Hota, Dongjoon Park, and André DeHon. Hipr: High-level partial reconfiguration for fast incremental fpga compilation. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 70–78. IEEE, 2022.

[109] Yuanlong Xiao, Eric Micallef, Andrew Butt, Matthew Hofmann, Marc Alston, Matthew Goldsmith, Andrew Merczynski-Hait, and André DeHon. Pld: Fast fpga compilation to make reconfigurable acceleration compatible with modern incremental refinement software development. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 933–945, 2022.

[110] Jinyi Xu, Kaixuan Li, and Yixiang Chen. Real-time task scheduling for fpga-based multicore systems with communication delay. *Microprocessors and Microsystems*, 90:104468, 2022.

[111] Siyuan Xu, Shuangnan Liu, Yidi Liu, Anushree Mahapatra, Mónica Villaverde, Félix Moreno, and Benjamin Carrion Schafer. Design space exploration of heterogeneous mpsocs with variable number of hardware accelerators. *Microprocessors and Microsystems*, 65:169–179, 2019.

[112] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019.

[113] Florian Zaruba, Fabian Schuiki, and Luca Benini. Manticore: A 4096-core risc-v chiplet architecture for ultraefficient floating-point computing. *IEEE Micro*, 41(2):36–42, 2021.

[114] Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads. *IEEE Transactions on Computers*, 2020.

[115] Qingxia Zhang, Zihao Meng, Xianwen Hong, Yuhao Zhan, Jia Liu, Jiabao Dong, Tian Bai, Junyu Niu, and M Jamal Deen. A survey on data center cooling systems: Technology, power consumption modeling and control strategy optimization. *Journal of Systems Architecture*, 119:102253, 2021.

[116] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. May 2020.

[117] Wei Zuo, Louis-Noel Pouchet, Andrey Ayupov, Taemin Kim, Chung-Wei Lin, Shinichi Shiraishi, and Deming Chen. Accurate high-level modeling and automated hardware/software co-design for effective soc design space exploration. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.