UNIVERSITÀ
DEGLI STUDI
FIRENZE

UNIVERSITÀ DI PISA

UNIVERSITÀ
DI SIENA
1240

PHD PROGRAM IN SMART COMPUTING
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE (DINFO)

A Data-Flow Threads Co-processor for MPSoC FPGA Clusters

Farnam Khalili Maybodi

Dissertation presented in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Smart Computing

THIS DISSERTATION IS CARRIED OUT AT THE UNIVERSITY OF SIENA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE E SCIENZE MATEMATICHE (DIISM)

A Data-Flow Threads Co-processor for MPSoC FPGA Clusters

ACADEMIC DISCIPLINE (SSD): ING-INF/05

Doctoral Candidate:

Farnam Khalili Maybodi

Advisor:

Prof. Roberto Giorgi

Head of the PhD Program:

Prof. Paolo Frasconi

Evaluation Committee:
Prof. Dionisios Pnevmatikatos, *The National Technical University of Athens*
Prof. Marco D. Santambrogio, *The Polytechnic University of Milan*

XXXIII ciclo — October 2021

To my parents

Acknowledgments

I would first like to thank my advisor, Prof. Roberto Giorgi, for his insight, vision, support, and scientific guidance over my Ph.D. program, and more for his patience with my technical (and non-technical) mistakes. Not to mention his original invitation to work in the European AXIOM project, without which I would not be where I am today. I am also grateful for being part of this project and working among experts in computer science and engineering domains, which were very rewarding for me.

I would also like to thank the members of my supervisory committee: Professor Antonio Prete and Professor Sandro Bartolini, for investing their time and effort and their helpful comments. Moreover, I am delighted to have Professor Dionisios Pnevmatikatos and Professor Marco D. Santambrogio as the reviewers for my thesis, and I am thankful for their valuable feedbacks. Also, I would like to thank the members of the evaluation committee, Professor Kavi, Professor Pnevmatikatos, and Professor Stea, for their constructive comments during my public defense.

Marco Procaccini, it has been a pleasure working with you for the past three years; you are a supporting friend to me despite your computer science skills and knowledge.

I am also thankful for the funding sources that made my Ph.D. work possible, including Region Toscana, the University of Florence, and the University of Siena, as well as the European Commission that partially funds the work presented in this thesis on AXIOM H2020 (id. 645496), TERAFLUX (id. 249013), HiPEAC (id. 871174).

Last but not least, I wish to thank my parents for their love and encouragement, without whom I would never have enjoyed so many opportunities. Their support and sacrifices to bring me and my sister up are priceless, and I cannot thank them enough for it.

Abstract

Data-Flow Threads (DF-Threads) is an execution model previously proposed by R. Giorgi [1] that distributes many asynchronous parallel threads in a multi-core multi-node architecture. These threads carry the input data to be processed, code pointer, and the output location where to store the results. In this model, program code is broken down into several threads, each of which is orchestrated by the Data-Flow Graph (DFG) across the processing elements. Each thread internally executes in a Control-Flow manner, while the distribution and the data-dependencies among the threads follow the DFG of the application. This hybrid setup of the computing execution harnesses fine-grain parallelism of DFG nature and the advantages of Control-Flow programmability.

This work presents architectural support for the DF-Threads execution model. For this purpose, we built a custom cluster of MPSoC FPGAs (i.e., AXIOM-board [2]) to explore the feasibility of the DF-threads model, and to demonstrate benefits or limitations. This thesis, for the first time, shows an implementation of the DF-Threads execution model on a reconfigurable hardware and proposes a modular architecture for a multi-node heterogeneous distributed system with network interconnections. Before implementing the hardware platform design, the model has been evaluated through our Design Space Exploration (DSE) tool-set in collaboration with my research group. Through the DSE, we noticed that this execution model has the potential to overstep programming models like OpenMPI. Then, the validated design is gradually migrated into the real-board tool-flow. We explain our methodology for evaluating an architecture in our DSE environment. We demonstrate the simplicity and rapidness of our methodology with a driving example that models a two-way associative cache. In the context of the AXIOM project, the DSE tool-set helped reduce the development time from days/weeks to minutes/hours.

We built the micro-architecture on the FPGA instance starting from a minimalistic API for supporting DF-Threads execution in previous works. The API allows us to handle the threads' lifetime, including creation, distribution, and destruction. The micro-architecture presented in this thesis is a DF-Threads Co-processor (DFC) fully implemented on the Programmable Logic (PL) of a Zynq Ultrascale+ FPGA (Xilinx). The proposed DFC is responsible for supporting the scheduling and distribution of the Data-Flow threads. The latter operations are entirely decoupled from the Processing System (PS), which only executes the DF-Threads. The DFC uses high-speed custom interconnects on the PL, through which it performs load-balancing of the threads between the nodes (the AXIOM boards). The load-balancing activity is advancing in parallel with the main computation, thus offloading the processor's burden for managing the parallel threads. The LBU is the central part of the DFC, and it is configurable in terms of parameters (e.g., fair-

ness, thresholds of the queues) to reach an optimum point in the total execution performance of the application.

We evaluate the implemented co-processor performance in managing the DF-Threads with simple benchmarks to stress the capability to manage many asynchronous threads. To our best knowledge, this is the first time to have two real nodes running DF-Threads, showing relative scalability matching the simulator's results. The results are reasonably aligned with the results obtained from the COTSon simulator and show the co-processor's capacity to manage and distribute many fine-grain threads in a multi-node heterogeneous architecture for real. Finally, we measured the essential voltage rails for exploring power consumption when performing high-speed board-to-board communications.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In the early 1940s, von-Neumann proposed a computer architecture, which is still widely used in modern computer organization. Based on this model, instructions are sequentially fetched and decoded by the Central Processing Unit (CPU), and the results are written back to memory. The evolution of digital computing was a significant milestone by the emergence of the first microprocessor (the processor structured on a single chip) by Intel under 4004 in 1971. This chip contained nearly 2300 transistors, and it was clocked at 740 kHz, executing 92000 instructions per second, while consuming around 0.5 watts. Since then, microprocessors had a massive advancement throughout their history. Figure 1.1 shows a 48-year of microprocessor trend data in terms of critical parameters (e.g., Frequency, power consumption, number of transistors) in the microprocessor evolution.

Single-chip microprocessors performance is directly proportional to the clock frequency, larger cache sizes and more complicated hardware mechanisms such as out-of-order and pipelining executions. However, there are limitations in improving performance due to the power wall and technology restrictions [3]. The designers moved the architectures from single-core to multi-core investing more in parallel computing, while the clock frequency remained constant (Since 2005, trend depicted in Figure 1.1). These cores typically operate with lower clock frequencies and are less power consuming than the conventional single-core microprocessors. However, in such multi-core architectures managing the cache data and memory hierarchy between the cores is an open research topic [4], [5]. We need to rethink execution and programming models to exploit properly today's complex multi-core systems ranging from shared memory systems to large-scale distributed memory clusters.

For this purpose, many efforts have been carried out in parallel-programming to exploit the available parallelism of many-core many-nodes computing organizations. For instance, Message Passing Interface (MPI) [6], OpenMP [7] and Cilk++ [8] for parallel execution models adopted in microprocessor, or domain-specific languages such

as CUDA and Map-Reduce are used in heterogeneous data streaming processing elements such as GPUs. Nonetheless, these techniques did not fully exploit parallel programming for a wide range of computing challenges (e.g., resource management challenges, communication bottlenecks, power efficiency) [9]. Moreover, the cost of synchronization in such parallel programming techniques scales up with the system size [10].

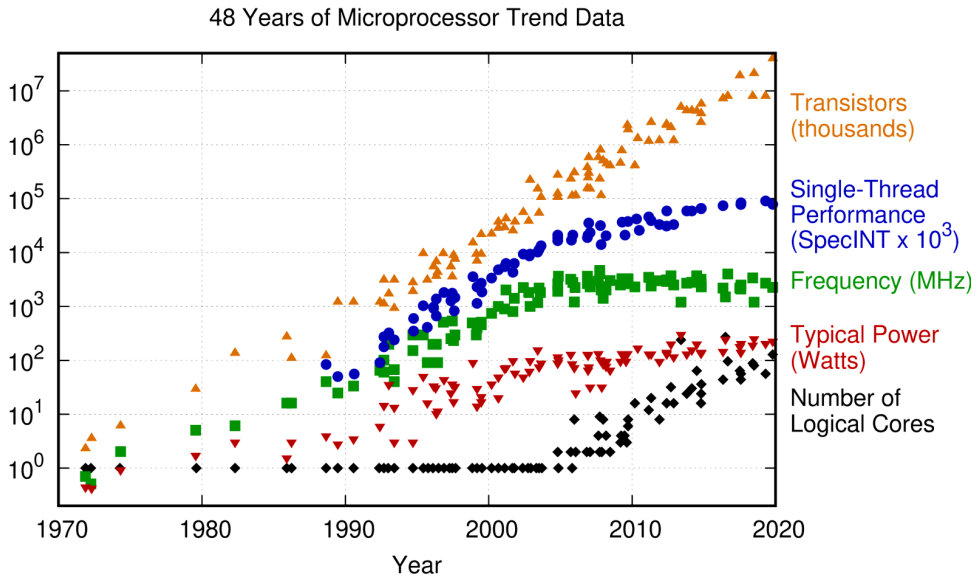

Figure 1.1: 48-years of microprocessor trend data showing the number of logical cores, typical power, frequency, single-thread performance and number of transistors [11].

The Data-Flow computing paradigm proposed by Jack Dennis [12], [13] is an alternative model for the von-Neumann computing model. This model inherently offers the potential of fine-grain execution and enables exploitations of parallelism while keeping the parallel programming synchronized due to its Data-Flow characteristics. The Data-Flow execution model is suitable on multi-core and many-node computing organizations by creating many asynchronous fine-grain threads orchestrated among the processing elements [14]–[27].

In the Data-Flow execution model, application codes (based on traditional programming models such as C) are translated into the Data-Flow Graph (DFG), representing the same execution graph of the application tasks. In DFGs, each node represents a function, alternatively, one or a set of instructions in line with the Data-Flow principle [13]. The data-dependencies between these nodes are represented by the edges (arcs) between the nodes. Each node's execution occurs when all of its inputs are available, in contrast with the von-Neumann models where the execution of an in-

struction starts once the program counter is reached to it, regardless the data execution order.

However, proposed designs with a pure Data-Flow machine was not totally promising. (e.g., Explicit Token Store architecture [28], and Manchester Data-Flow Machine [29]), mainly because of the excessive fine granularity at the instruction level and poor capability to effectively manage the data structures and memory systems yet respecting conventional programming languages [30]. To cope with this issue, a possible solution can be a hybrid design of the von-Neumann and Data-Flow models of computation [30], [27], [31].

This study mainly focuses on implementing a Data-Flow execution model so-called DF-Threads [1], on a multi-board heterogeneous reconfigurable platform (the AXIOM board) and validating the performance with a simulator framework (COTSon).

The preliminary work required the development of a methodology and tool-set as part of the Design-Space Exploration (DSE), which enhances the spent time of hardware prototyping and performance evaluation of the execution model. The AXIOM project [32]–[36], [2], [37]–[39] aims at building a software/hardware environment suitable for heterogeneous embedded system in the Cyber-Physical Systems (CPS) domain. We take this opportunity to test and verify our proposed execution model's functionality for real-world applications covered by the AXIOM project. The proposed design of the DF-Threads allows offloading the fine-grain threads management to a co-processor so-called DF-Threads Co-processor (DFC), which leads to an acceleration of function execution and thread orchestration among the available multi-core nodes of the organization. Although the DFC can be implemented in software, we implement it entirely on the Programmable Logic (PL) part of the FPGA Zynq Ultrascale+ chip of the AXIOM board, which reduce the significant processing burden of local ARM cores, which might be dedicated to the DF-Threads management across the FPGA board cluster. The AXIOM board has the following key features: i) several low power cores ii) a high speed reconfigurable interconnect for board-to-board communication; and iii) a user-friendly programmable environment, which allows us to offload program algorithms partially into accelerators (on programmable logic) and, at the same time, to distribute the computation workloads across boards (see Figure 1.2).
During the development of the design, the DF-Threads execution model was previously validated and evaluated by the Design Space Exploration (DSE) tool-set (named MYDSE) [40], leading to saving not only the development time but also numerically evaluate the possible bottlenecks of the proposed model before migrating it on the AXIOM platform. Thus, moving an already-validated design from the simulator (COTSon [41]) in terms of functionality and performance to real hardware facilitates the design development.

Two benchmarks (Matrix Multiplication and Recursive Fibonacci) have been selected to gauge the DF-Threads execution model's performance. Benchmarks like Matrix Multiplications are frequently used in computational demanding real-life applications such as Smart Video Surveillance (SVS) [35] and Smart Home Living (SHL) [36]. These applications consume a significant compute power to analyze many scenes from multiple cameras, e.g., airports, homes, hotels, and shopping malls. We use the Recursive Fibonacci benchmark (a fine-grain algorithm) as a preliminary stress test of our proposed implemented co-processor on a cluster of FPGAs. We verify and evaluate the deployed load balancing technique's performance and the memory management across the SoC FPGAs with this benchmark.

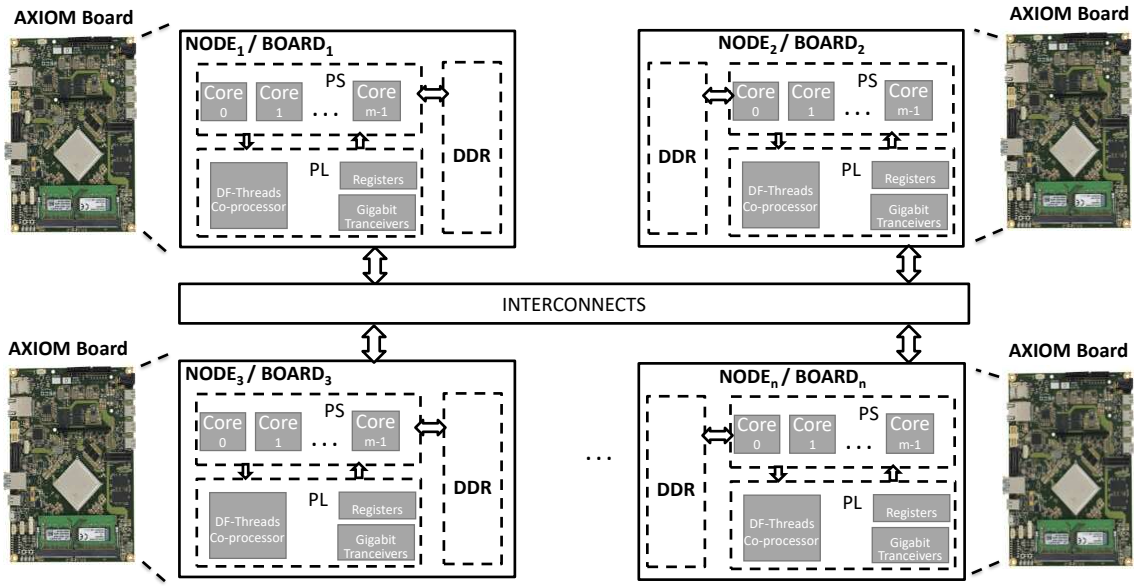

Figure 1.2: Architecture of the distributed system based on the AXIOM boards. The Distributed System consists of N Nodes based on an MPSoC, which includes a Processing System (PS) and Programmable Logic (PL). The nodes of the system are connected through USB-C cables without the need for an external switch.

1.1 Problem statement

The trend of computing growth encounters the end of Dennard scaling and a slowed down Moore's law while still increasing the number of transistors. It is due to elevating parallelism exposed by multi-core architectures to achieve high-performance [9]. These cores are based on Control flow processors (von-Neumann), on which parallelism has been applied on different levels; thread-level parallelism (TLP) (programmer dependent), data-level parallelism (DLP) (compiler and programmer dependent), and instruction-level parallelism (ILP) (hardware dependent).

There are several proposed programming models like OpenMP [7], MPI [6], and Cilk++ [8] to program such multi-core systems. Although these programming models may facilitate parallel programming, they have synchronization limitations that arise from waits due to data-dependent long latencies memory accesses. It leads to low resource utilization and performance inefficiency, mostly in a distributed shared memory and multi-core many-node systems [42]. Thus, we need to rethink such programming and execution models to fully exploit parallelism offered by on-chip resources and achieve a better performance and power efficiency. However, even with full exploitations of available parallel resources, the overall performance is limited by the sequential portion of a task, as Amdahl's revealed [43], [44]. Moreover, real-life applications running on current multi-core clusters usually exploits up to 10% of the peak processing power of the system, and encounters CPU starvation [45].

Regardless of the execution model, several programming models are proposed that offer flexibility and high-performance capability, such as task-based programming models [46]–[48]. These software models take advantage of the offered parallel granularity of Data-Flow models and improve multi-core architectures source utilization. In these programming models, the programmer does not deal with the explicit management of the parallelism, while the scheduler is responsible to implicitly schedule data and computations of tasks. However, the introduced processing power overhead to schedule and manage the parallel tasks (especially small tasks) significantly affects the overall performance efficiency of the system.

Data-Flow execution models stand as an alternative to the von-Neumann computing model [13]. However, the pure Data-Flow architectures have limitations due to their inability to support the imperative programming languages and data structures. Moreover, due to some limitations of the Data-Flow execution model, they cannot be considered an alternative to traditional general-purpose processors (GPPs). For instance, control transfer could be expensive in some Data-Flow models, or the latency costs of direct data communications can be restrictive [49]. The problems mentioned above made us rethink alternative solutions to utilize the resources of multi-core architectures efficiently.

A hybrid model is presented in this work to exploit the full parallelism offered by heterogeneous multi-core/multi-node, where the threads are scheduled following the Data-Flow model, and the instructions are executed in a von-Neumann model. In this model, exploiting parallelism consists of breaking an application code into several DF-Threads (each of which composed of one or several instructions to be executed in a GPP core) and coordinating them for their correct execution. This model has a Data-Flow management co-processor, which controls and manages the execution of the Data-Flow Threads (DF-Threads) [1]. This co-processor's main component is the load balancing unit, which takes care of the asynchronous distribution of the DF-Threads among the available computing elements (i.e., local cores and remote nodes). Each DF-Threads executes by a GPP core, while the orchestrations of the DF-Threads among the available cores are based on the Data-Flow model. This solution offers many advantages against either pure von-Neumann and pure Data-Flow models.

The advantages of our DF-Threads as listed below:

i) No need for memory consistency due to the Data-Flow graph's inherent data consistency; this provides a power-efficient and straightforward design that can effectively exploit the processing power of available cores.

ii) Asynchronous execution of fine-grain DF-Threads [50], which provides a great potential to effectively utilize the parallelism offered by the multi-core multi-node architecture.

iii) The length of the independent data path is the expression of the granularity of parallelism. The granularity of the DF-Threads can be adjusted from fine-grain to coarse-grain; this makes the DF-Threads execution model a flexible computing model covering the optimum execution of any applications.

iv) The idea is applicable either on software or on hardware (ASIC or FPGA).

1.2 Goals and challenges

Energy efficiency in current and future High-Performance Computing (HPC) systems is a fundamental challenge that cannot be approached by scaling the number of cores in a multi-core architecture. Therefore, one possible solution can be heterogeneous architectures. Recently, FPGAs have proven to be reliable and energy-efficient platforms, thanks to their reconfigurable spatial infrastructure. Given the previous motivations, this thesis focuses on implementing a novel execution model based on the Data-Flow threads (DF-Threads), [51], [52], on a distributed heterogeneous computing platform. We show the functionality of the DF-Threads execution model on an MPSoC FPGA

Cluster. Furthermore, we study the capability of full exploitation of parallelism in a multi-core/multi-node heterogeneous architecture.

In this study, we try to show our novel execution model's functionality on a cluster of heterogeneous reconfigurable platforms, which offer scalability and the potential to a better performance and power efficiency [53], [54].

We necessitate the following goals and challenges to have the DF-Threads execution model running on a distributed heterogeneous reconfigurable system:

- The extension of the DF-Threads API with the integration of the DF-Threads Co-processor (DFC) as the accelerator implemented besides the CPU.
- Efficiently exploitation of parallelism: the DF-Threads execution model distributes the threads which are ready to be executed across the available nodes in a cluster of MPSoC FPGAs (e.g., AXIOM board). For this goal, we deploy a load-balancing module, which dynamically orchestrates the DF-Threads across the cluster, without intervening of the user API.
- Performance analysis for the SoC FPGA implemented DFC and validating the results with the simulator (COTSOn simulator).

For this purpose, thanks to the AXIOM project, we have a scalable, reconfigurable, and distributed platform as the hardware infrastructure of our proposed execution model. Our proposed architecture is implemented on the Programmable Logic (PL) of an SoC FPGA (AXIOM board). The FPGA of the AXIOM board is a heterogeneous architecture based on a Zynq Ultrascale+ composed of Processing System (PS) and Programmable Logic (PL). PS cores comprise four Cortex A53 ARM cores that allow us to be suitable for a large set of applications, and the PL is known for its power efficiency and reconfigurability. It makes it an appropriate choice for being used in the multi-threads Data-Flow execution models. These models evolve around the data mobility optimizations and massive exploitations of parallelism among many thousands of DF-Threads, which offers higher performance and modularity [1], [23], [55], [56].

The DFC comprises a scalable thread scheduler unit that is responsible for the data synchronization and DF-Threads distribution among the PS cores of multiple heterogeneous boards, exploiting the computational power of the PS cores only for the execution of the threads. The design first is verified, and the performance of the DF-Thread execution model is evaluated through our DSE-tool-set, and COTSon simulator [40], then migrate the design on the AXIOM ecosystem, which offers a cluster of a heterogeneous system.

1.3 Contributions

These are contributions of this thesis:

1. FPGA prototyping of a preliminary design of DF-Threads execution model. The design previously was simulated through our design space exploration (DSE) tool-set [40] in our research group (see Chapter 3).

2. Performance validation of results obtained through the utilization of simulator (COTSon Simulator) and a real board (The AXIOM board) based on an FPGA prototyped model running a benchmark (i.e., Blocked Matrix Multiplications)[39] (see Chapter 3).

3. Translating timing model into the High-Level Synthesis (HLS), and generalizing the methodology to the AXIOM project (Zynq Ultrascale+ board) [57] (see Chapter 4).

4. Proposing architectural support for the DF-Threads execution model, which is implemented for the first time as a co-processor (DFC) on a real hardware based on MPSoC FPGA (Zynq Ultrascale+). The major component of the DFC is the load balancing unit (LBU), which orchestrates DF-Threads across the available nodes throughout the cluster (see Chapter 5).

5. Analysis of the power consumption and key performance metrics of the target platform (Zynq Ultrascale+) for being used in a multi-node computing platform (board-to-board high speed communication) [58] (see Chapter 5).

1.4 Thesis outline

The work presented in this study is organized as follows: Chapter 2 addresses the related work to this study. In Chapter 3, the DF-Threads execution model, the Design Space Exploration (DSE) tool-set, and the COTSon simulators are briefly introduced. The performance validation of the DF-threads execution model with a preliminary design on an SoC FPGA based board (AXIOM board) is presented. In Chapter 4, the methodology adopted to translate the timing model of the architecture into the High-Level Synthesis (HLS) tool is detailed. The methodology is demonstrated through a simple case study example consisting of modeling a two-way associative cache. Chapter 5 presents the proposed architectural support for the DF-Threads execution model, targeting a cluster of SoC FPGAs. The proposed design is implemented on the AXIOM board, and the analysis of power consumption and important key metrics is performed. Chapter 6 concludes this thesis and addresses possible optimizations and directions to future work.

Chapter 2

Background Knowledge

Nowadays Field Programmable Gate Arrays (FPGAs) have become a promising candidate for High-Performance Computing for their offered scalability, reconfigurability and power efficiency. This study aims to materialize an execution model capable of parallelizing threads execution across the available computing resources while keeping the simplicity of the von-Neumann-based execution model. To have this model functioning, we take advantage of the MPSoC FPGA architecture (i.e., Zynq Ultrascale+). To cover the related work to this study, first, we describe the Data-Flow computing paradigm proposed by Jack Dennis [13], as an alternative to the von-Neumann model of execution. Next, we briefly highlight the different types of Data-Flow computing.

2.1 Data-Flow computing models

The Data-Flow computing paradigm is one of the possible alternatives to the von-Neumann computing paradigm, and a computer architecture based on this model was presented by J.Dennis [59], [60] and G. Kahn [61] in the early 1970s. In the Data-Flow model, a computing problem breakdowns into the Data-Flow nodes and arcs of a Data Flow Graph (DFG) [62], [14]. The nodes represent the program's instructions, and the arcs (edges) represent the data-dependencies between the instructions. In this model, each node becomes executable once the data of all its inputs are available. In this context, the principle of Data-Flow allows the instructions to be executed in parallel, which offers the potential for increasing the Instruction Level Parallelism (ILP) at the instruction level [63]. Moreover, the Data-Flow model is one of the major methodologies to map the high-level languages onto the digital hardware. The Data-Flow model has been extensively used in many digital computing research areas, including graphics processing, data warehousing, and digital signal processing, and machine learning [64], [65].

Data-Flow models can be implemented both in software and hardware. The software ones are usually part of runtime libraries or virtual machines [47], [66]–[68]. On contrary, the hardware implementations of Data-Flow model targets Application-Specific Integrated Circuits (ASICs) [69]–[71], or Field Programmable Gate Arrays (FPGAs) [72]–[75].

Data-Flow execution models have been reviewed recently [1], [30], [76] as they promise an elegant way to efficiently execute threads based on data availability [77], [27], [78]. Notably, the computations can be mostly performed in a producer-consumer manner, while for mutable shared data, the memory model offered by Data-Flow Threads (DF-Threads) [1] is enclosing Transactional Memory [79], which is a concurrency control mechanism for controlling access to shared memory by replacing locks with atomic execution units.

In this context, the TERAFLUX project [80], [51], [15], [81], [52] accomplished such Data-Flow modality while extending to multiple nodes which are executing seamlessly through an appropriate memory semantic [82], [1]. A compound of consumer-producer patterns [52], [83] and transactional memory [84], [79] allows a novel combination of Data-Flow paradigm and transactions to solve the consistency issues across nodes, where each node is supposed to be cache-coherent like in a classical multi-core. Additionally, such distributed systems could support fault-tolerance [85], [86], and in this context, a Data-Flow thread may be re-executed without harming the computing program since the thread inputs are maintained before scheduling the corresponding thread. The Data-Flow concept is also extensively used in software as the semantic of programming language modeling concurrent tasks and using the parallelism offered by the Data-Flow machine.

There are various levels of computing abstractions, in which the Data-Flow concept can be deployed from a pure Data-Flow hardware implementation to the Data-Flow at the instruction level, thread-level, or hybrid Data-Flow architectures [87]. The classic Data-Flow architectures can be classified into two categories, static Data-Flow and dynamic Data-Flow. In the following, the variations of this model are highlighted and addressed.

2.1.1 Static Data-Flow architectures

The static Data-Flow model (The first model proposed by J. Dennis [60]) allows only one instance of a node to become ready to execute. Moreover, at most only one token resides on any arc. Indeed, a Data-Flow node can be executed when all of its input tokens are available, and no tokens reside on any of its output arcs. In this model, the

memory collection comprises three types of cells, including opcode, destinations address(es), and operands. Each functional node in the DFG consists of five pipelined functional units, which decodes the op code and performs operations, encapsulates the output packet, and writes back the result tokens to the memory cells. The enable nodes are inherently detected. However, the graph is static, and every instruction can be instantiated only one time, which limits the performance of loop iterations and subprogram invocations. The main factor that is responsible for moving the results from the processing elements to the memory is a distribution network module. In order to reduce the overhead of the network, only the acknowledgment signals and boolean tokens are exchanged between the processing elements and the memory cells. To avoid indeterminacy, some extra arcs are dedicated for acknowledgment signals from the consuming node (the node that asks for input tokens) and producing node (the node that its output arcs carry valid results). These acknowledgment signals ensure that no arc will carry more than one token. A token on an acknowledgment arc indicates that the corresponding data arc is empty. When a node executed the instructions, it asserts an acknowledge signal to receive the next token (the acknowledge signals follow the Data-Flow chain of the program).

However, this model suffers from essential issues that are listed below:

- Low performance in loop iterations and re-entrancy [88]
- Considerable network overhead due to the acknowledgments and tokens
- It lacks the general support for programming constructs in imperative programming languages (like recursions and procedure calls)

However, despite the disadvantages of static Data-Flow architectures, some machines are designed based on the static Data-Flow concept. The most important static Data-Flow architectures are: Hughes Data Flow Multiprocessor (HDFM) [89], MIT Data-Flow architecture by J. Dennis [60], [90], Data Driven Machine1 (DDM1) [91], Language Assignation Unique (LAU) system architecture [92], and the TI Distributed Data Processor (TDDP) [93].

2.1.2 Dynamic Data-Flow architectures

The dynamic Data-Flow model solves some of the limitations of static Data-Flow, such as low performance introduced by one instance execution per time for loop iterations. Thereby, dynamic Data-Flow supports the concurrent execution of multiple instances of loop iterations at runtime, which improves the machine's performance. It is done by viewing arcs as buffers containing multiple data items. Dynamic Data-Flow models

are so-called tagged-token Data-Flow models, due to the associated tag with each token. A tag representing the dynamic instances is assigned to each token to distinguish among instances of a node to identify the context in which a specific token was generated. An actor starts firing once all of its input tokens with identical tags are available. This firing rule in dynamic Data-Flow models removes acknowledgment signals, leading to reduced overhead.

A notable example of dynamic Data-Flow architecture is Manchester Data-Flow Machine [29], [69]. The achieved performance for this machine is up to 1.2 MIPS (Million Instructions Per Seconds). This machine composes of five units organized as a pipeline ring (see Figure 2.1). These units are listed below:

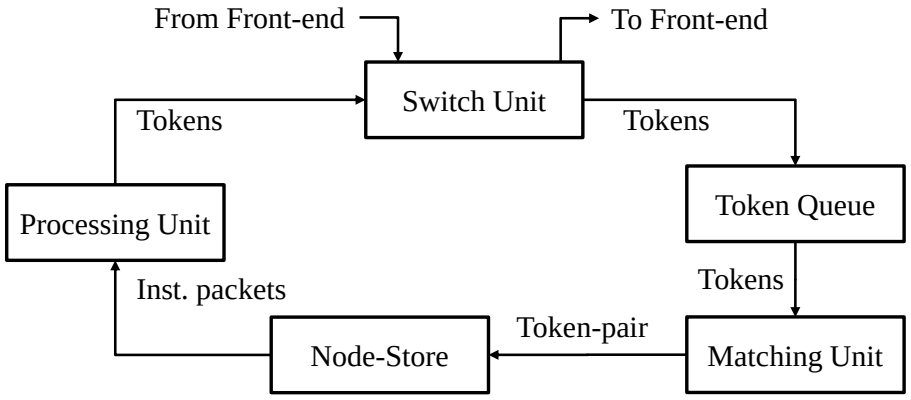

Figure 2.1: Manchester data-flow five stage pipeline ring [87].

- I/O Switch: it establishes communications between the host and Data-Flow Processor Unit (PU). This switch allows programs and data to be loaded from a host processor and results to be written back to the host for external inspection.

- Token queue: this is the FIFO (first-in-first-out) queue to smooth out uneven rates of the traversing tokens throughput across the pipeline ring.

- Matching unit: this contains six pipeline registers, a parallel hash table, and a 16-bit interface to the over-flow unit. Each hash table consists of a 64 Ktoken memory in addition to a 54-bit tag/destination comparator and interface control. The matching unit is responsible for collecting tokens with identical tags by pairing tokens accordingly with the same destination node addresses. Over-flow takes place when the accessed locations are entirely occupied. In this case, the non-matched received token is passed to the over-flow unit with the associated indicator flag.

- Instruction store (Node Store): this consists of two pipeline registers, a segment lookup table, and a random-access memory to store instructions of the program.
- Processing Unit: it is a micro-programmed, 2-stage pipeline unit that executes the Data-Flow operations. The first stage moderates the generation of results tokens with their associated tags, and the second stage comprises 15 functional units to execute the instructions operations.

There are other machines following the baseline of the dynamic Data-Flow architectures, such as PIM-D [94], the Tagged-Token Data-Flow Architecture (TDDA) [95], Distributed Data-Driven Processor (DDDP) [96], and SIGMA-1 [97].

Despite the more parallelism that dynamic Data-Flow offers, it suffers from several drawbacks [88], [87]. One of these disadvantages is the extra imposed overhead for matching the tags on tokens. The authors in [69] propose a method to mitigate the overheads coming from the match process at the expense of the more complex memory implementations. However, the overhead from matching tokens and prohibitive costs of associative tag lookups yet introduce a considerable limitation in terms of silicon area, latency, and power efficiency. The shortcomings in implementations of the dynamic Data-Flow are listed as below [87]:

- Resource allocation is complex
- Matching tokens with associated tags impose a considerable overhead.
- The instruction cycle is not handled efficiently due to non-matching deadlocks
- Data structures are not too easy to be managed by the programmer
- Performance is strongly dependent on the rate at which the matching unit operates

Explicit Token Store (EST) tries to overcome the associated overheads with the token store coming from inefficient matching process [98]. One of the significant issues in dynamic Data-Flow architectures is the efficient matching process. To overcome the inefficiency coming from the matching of the dynamic Data-Flow model, ETS offers a direct matching. The main idea relies on the dynamic allocation of an independent memory frame, the so-called activation frame for all tokens generated by code block (e.g., each activation of a loop iteration or subprogram invocation). These allocated frame stores are associated with the frame of wait-match storage on each code-block invocations and hold synchronization information of instructions within the code-block. An offset relative to a pointer is associated with the frame through which access to locations within the activation frame is possible, leading to a fast and explicit addressable memory, thus eliminates the need for associative memory searches. The runtime performs

the allocation procedure. The traversing token comprises a continuation pair <FP, IP> Instruction Pointer (IP) (also called destination instruction), and Frame Pointer (FP) to activation frame. Typically, an instruction carries an opcode (e.g., ADD), an offset from the activation frame pointer, where a match will occur, and displacement fields specifying destination instructions that receive the output tokens. Moreover, the number of the input port of each destinated actor is determined by the displacement fields. Once the continuation pair arrives by the Processing Element (PE), the instruction is fetched from the Instruction Memory pointed by IP, then FP with the offset exactly points to a cell within the wait-match memory. If it was empty, the arrived continuation (token) should be stored in the Frame Storm; if the slot is full, the partner token is retrieved then the instruction will be dispatched. A notable examples of using ETS architecture are Monsoon architecture [70], and Epsilon-2 multiprocessor [71].

2.1.4 Data-Flow architectures' general limitations

Despite the massively exposed parallelism of Data-Flow architectures, they still suffer from several issues. One example is that since no concept of a variable exists and data flows under the tokens among the instructions, arrays and data structures can not be handled easily as in a control flow machine. Moreover, a sequential code consisting of loops, structures, arrays, and recursions, still may need several considerations to be accomplished for the compiler, and programming requires a complex way of thinking yet. Although the Data-Flow concept works well for traversed variables under the tokens, passing a complex data structure and handling it among the nodes may not be trivial. A data structure type should be passed under the token, and any update to any cell of structure leads to a new token. Moreover, respecting the single-assignment rule (values associated with variables cannot be modified) when a token point to a data structure, the cells of the structure should not be modified. Another general issue in Data-Flow architecture is an extra overhead due to handling the token per instructions and fine-grain context switching at each instruction level. The pipeline is not utilized for the same threads executing instruction when previous instructions are not yet executed in the architecture.

In practice, more efficient management of structures and arrays are still missing. However, there are proposed solutions to mitigate these issue such as direct access method [99] and indirect access method) [100].

Data-Flow architectures are specialized for functional languages and not welcoming to general programming models widely used in high-performance computing. For this reason, particular programming languages are needed (see Section 2.1.6), which produce large DFG to exploit as much as possible the exposed parallelism of the underlying architecture. However, such programming models may have several issues

in handling explicit computation states with complex data structures and loops. To overcome the complexity of data structure, some solutions are proposed as specialized storage mechanisms such as the I-structure [101], and still respect single-assignment. However, these modifications complicate the design, directing the data structure to a non-generic structure. Another issue in such a machine's programming is its inability to execute instructions widely used under the memory ordering by imperative languages (such as C/C++). In general, pure Data-Flow architectures (as discussed in previous sections) are not welcoming and effectively supporting imperative languages (e.e., C/C++).

Consequently, the Data-Flow architecture is one solution to exploit parallelism, coming from the explicit expression concurrency across the Data-Flow graph paths, thus splitting the execution rather than centralized execution of instructions offered by the Control-Flow model. However, as far as the granularity of this parallelism introduces difficulties in handling imperative programming concepts and, hence limits the effectiveness of such machines [102]–[104].

2.1.5 Hybrid Data-Flow/von-Neumann execution models

The hybrid architectures attempt to mitigate shortcoming of classic pure Data-Flow architectures (e.g., static and dynamic) by combining the concept of Data-Flow with Control-Flow execution models [105], [30]. These models benefit either the parallelism and inherent synchronization of Data-Flow and the programming abstraction and support based on von-Neumann models by combining the abstractions of the Control-Flow and shared data structures. For example, many hybrid models take advantage of Data-Flow scheduling techniques and programming based on Control-Flow approaches. In the hybrid model, a node of a DFG comprises a set of sequentially executable instructions as a so-called thread of instructions. As a result, based on this feature, the granularity of executable instructions may vary from fine-grain (an instruction) to coarse-grain (a set of instructions).

A further benefit of hybrid architecture relies on their memory models, still integrating the single-assignment semantic of Data-Flow nature while owns memory consistency of Control-Flow execution. It alleviates the problem of inefficient support of shared data structures and arrays, thereby managing imperative languages.

A further implementation of DF-Threads [106], [1] discussed in this thesis relies on the hybrid model, which maps the application into many threads forming a DFG. These threads among the DFG execute on Control-Flow cores.

Hybrid models can be classified into a wide range of granularity in several operations, from a single instruction (fine-grain) to a set of instructions (coarse-grain). The Out-of-Order model [107], which is an extension of the superscalar processors. In this work, the Data-Flow semantic is deployed into the issue and dispatch stages to enhance ILP (e.g., local Data-Flow or micro Data-Flow architecture [108], [109]), and power efficiency. These models exploit more efficient ILP than the pure von-Neumann models. However, due to the limitation in the silicon area and technology, their parallelism is particularly restricted. As other notable examples of hybrid model, the MIT Hybrid machine [110], , the EM-4 architecture [111], the Epsilon-2 multiprocessor [71], the McGill Data-Flow Architecture (MDFA) architecture [105], the memory Decoupled Threaded Architecture (DTA) [112]. A further implementation of hybrid models is: TRIPS [113] TARTAN [114] and DySER [115], which execute a set of instructions under a block in a Data-Flow semantic, while each block is scheduled in a Control-Flow way. In contrast, the DF-Threads execution model [1] schedules the blocks of threads in Data-Flow form and executes each instruction included into the blocks based-on Control-Flow semantic.

Further examples for hybrid model are Star-T (*T) [116], TAM [117], ADARC [118], EARTH [101], MT. Monsoon [105], Plebbes [119], SDF [103], DDM [120], Carbon [121], and Task Superscalar [122].

2.1.6 Data-Flow programming languages

The Data-Flow languages typically use some Data-Flow principles, including single assignments, based on which values assigned to variables remain unchanged, and locality of effect, based on which instructions do not have unnecessary data dependencies. Due to scheduling determined from the dependencies, the value of variables does not change between their use. It is the only way to guarantee the avoidance of the reassignment of variables once their value has been assigned. A benefit of the single-assignment is that the order of statements is irrelevant. There is no circular reference, and each variable's creation can occur in any order during the execution. However, in these languages, the order of the statements is essential when a loop is being defined. To respect to the single-assignment rule, an imperative syntax is used to determine the value of the variables on the next iteration of the loop (e.g., ***next*** in [123]). A Data-Flow program does not allow a function to edit its own parameters to guarantee the validity of data dependencies, which can be avoided by the single-assignment rule. Consequently, Data-Flow programming languages are extensively functional languages using imperative syntax [124].

Moreover, there are a few Data-Flow programming languages that support recursion instead of loops [125]. Nonetheless, it is important to be noted that the proposed

DF-Threads [1], [106] can execute either loop and recursions (e.g., Recursive Fibonacci benchmark).

Value-oriented Algorithmic Language (VAL) is proposed by MIT [126], and is based on a textual representation of Data Flow Graph (DFG), and relies on pure functional language semantics. VAL offers an implicit concurrency, which means that the operations execute simultaneously without the need for any explicit language notations. The language uses expression and function-based features, which prohibits the side effects simplifying the translations to the DFG.

ID (Irvine Data-Flow language) [127], which was proposed by Arvind for the dynamic Data-Flow architecture permits high-level programming language based on block-structured, expression-oriented, and single assignment principles. It has an interpreter to translate and execute the programs for the dynamic Data-Flow machine. The types of variables are declared implicitly by the values that they carry. Each structure is an array of elements, which can be accessed through the indices or string values. It uses two syntaxes to work with data structures, ***append***, ***select***, and ***new*** ; ***append*** is used to create a new structure by duplicating the elements of the original structure, and ***select*** is used to read the value of an element of the structure. It uses ***new*** to define a new value associated with an expression.

Data flow Language (DFL) [128], which is based on the Petri nets [129] and nested relational calculus (NRC) [130] is a graphical workflow language. In this language, a set of complex data is defined to describe the structured complex values modified by the workflow. Then, the structure of this data is reflected in the structure of the workflow. The language has a formal semantics that is based on the Petri nets and NRC to support both the control flow and Data-Flow models. They extend the Petri nets to reorganize the processing tasks, and NRC (it is a query language over complex objects used in database management) to handle a set of data items such as those that are used in "for iteration" and "typing system". In DFL, the Data-Flows are constructed in a hierarchical manner, based on a set of refinement rules, and are initiated with a single token in the input node and terminated with another single token in the output node. It reflects the result at the output eventually computed in any case, without considering how the computations proceed.

Streams and Iterations in a Single Assignment (SISAL) is a derivative of the Val [126]. It is based on texted functional DFL, which was coined by Feo and Cann [131]. The syntax of the language is similar to Pascal and is strongly-typed, with optimized readability and learning curves. It also provides a micro-tasking ecosystem, which executes the program based on the Data-Flow model on the conventional single-core machines. Its compiler can distribute workloads among nodes in an optimized way in

a fully automatic manner. The compiler is responsible for sketching the DFG and create the nodes and arcs before the runtime. In runtime, the nodes are executed in parallel, and the data from one node to another is forwarded along the Data-Flow chain.

The Textual Data-Flow Language (TDFL) was proposed by Weng [125] as one of the oldest developed Data-Flow languages. The Data-Flow graph is generated with data streams in a straightforward manner during the compilation phase. The program is broken down into a series of modules, each of which comprises a set of statements that are assignment (based on the single-assignment rule), conditional statements, or call to other modules. Each module can call itself recursively, respecting the single-assignment rule while the iteration is not provided directly.

Language Assignation Unique (LAU) language is proposed by Computer Structures Group of ONERA-CERT in France [132] for use in static Data-Flow architecture. It includes conditional branching and loops that are adapted with the single-assignment rule through the use of old keywords. It provides explicit parallelism through the expanded keyword specifying concurrent assignment. This language has some specifications similar to object-oriented languages such as data and operations encapsulation.

LAPSE [133], which was derived from Pascal and proposed being used on the Manchester Data-Flow machine [29]. Similar to other DFL (Data flow languages), it is based on single-assignment semantics and offers conditional evaluation, user-defined data types, and functions. In contrast with LAU and ID, it does not use any qualification keywords to distinguish the current and next value of the loop variables in iterations. Moreover, its compiler assigned the old value of a variable if it has appeared in an expression, and the next value will be assigned when it appears on the left of an assignment. LAPSE offers a single explicit parallel construct similar to LAU language for all the concurrent assignments.

Lucid [123] is a functional language that enables formal proofs independently of the area of Data-Flow. It was realized that the iteration has two non-mathematical specifications, which are transfer and assignment. In this language, statements are simply axioms that proceed the proof by traditional logical reasoning using a few axioms and rules of inference for the specific functions defined in Lucid. Lucid programming is quite similar to a conventional imperative programming language with an assignment, conditionals, and while loops. However, Lucid realizes the assignment statements as equations, and loops are created implicitly by introducing restrictive assignment. Although, it is intended to be designed out of Data-Flow domain, its functional and single-assignment semantics made it similar to those required for the Data-Flow architectures.

Chapter 3

Design Space Exploration (DSE) tool-set

In order to match the performance request with the design requirement, researchers more than ever rely on the heterogeneous and domain-specific architectures [134]. Heterogeneous architectures may be composed of thousands of tightly coupled cores (CPUs and GPUs), residing nearby accelerators, and become more complex than current ones [135]. Moreover, modern embedded systems are increasingly based on heterogeneous Multi-Processor SoC (MPSoC) architectures to reach better performance and eneregy efficiency. To cope with the design complexities of such architectures, Design Space Exploration (DSE) is an important portion of the design flow. DSE and its automation is a significant part of modern performance evaluation and estimation methodologies in order to reduce the design complexity, time-to-market, and find the best compromise among design constraints in respect to the application.

Computer designers, therefore, rely on simulations as part of the DSE work-flow to perform early-stage design assessments in order to save time and costs. A simulator not only ensures the functional correctness but also may provide an accurate timing information.

Evaluation of a multi-core architecture even at the prototype stage is quite challenging, time-consuming. Moreover, it is not always possible to get the ***perfect*** setup, and hardware prototyping possibly imposes several limitations. In this section, we briefly highlight these limitations and show the importance of a simulator (like COTSon [41]) when it comes to assess and retrieve key metrics of a high- performance computer, e.g., 1000 general purpose cores.

We use COTSon (see Section 3.1.1) to offer a flexible DSE tool-set that easily can adopt new hardware/software platforms, and support scalability for a multi-node architecture. For instance, in order to address the challenges of a 1k-core architecture, we should be able to have a full-system simulation including Operating System (OS),

application benchmarks, a memory hierarchy and all peripherals as well.

3.1 Simulation framework

Our designed framework, which has been developed in our research group allows us to modify system parameters such as the number of cores and number of simulated instances (nodes), which are running in parallel on completely independent guest virtual machines. This framework can support different programming models such as OpenMPI, Cilk, or Jump [106].

The simulation framework relies on HP-Labs COTSon simulation environment [41] and on a set of customized tools that are intended to easily setup the experimental environment, run experiments, extract and analyze the results [40].

3.1.1 COTSon simulator

COTSon simulator [41] is based on the so-called ***functional-directed*** approach, where the functional execution is decoupled from the timing feedback. COTSon simulator uses the AMD SimNow virtualizer tool, which is proposed by AMD in order to test and develop their processors and platforms. COTSon executes its functional model into the SimNow virtual machine, running and testing the execution of the functional model. A custom interface is provided, in order to facilitate the exchange of the data between COTSon and the internal state of SimNow. As can be seen in Figure 3.1, COTSon architecture is made of three main components:

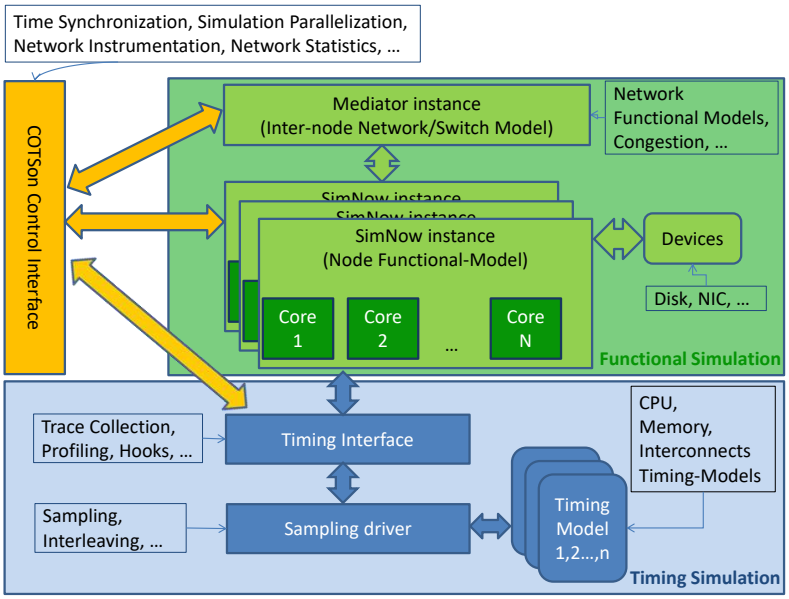

Figure 3.1: The COTSon simulation framework architecture [40], [136].

1) FUNCTIONAL MODELS: it contains the instances of the SimNow virtualizer, which executes the functional model based on a configurable x86-64 dynamically translating instruction-level platform simulator.
   In fact, we were able to customize the x86-64 instruction set of SimNow in order to introduce new instructions for the implementation of the DF-Threads execution model [52].

2) TIMING MODELS: it implements simulation acceleration techniques, such as dynamic sampling, the tracing, profiling and statistics collection. Through the specification of a timing model for a given component (i.e., L1 cache, networking), we can model different behaviors. The timing models are decoupled from the functional execution of SimNow, allowing us the flexibility to model different types of architectural feature.

3) SCRIPTING GLUE: the final part is related to the scripts used to boot/resume/stop each virtual machine, the setup of the parallel simulation instances of SimNow and the time synchronization among all the virtual machines.

3.1.2 DSE tools

In order to guarantee a proper scientific methodology for experimentation, we developed the Design Space and Exploration Tools (DSE Tools) through which is possible to easily set up the COTSon simulation environment, extract and analyze the results of the experiments.

The normal tool-flow is based on the next eight steps.

i) GENIMAGE: the SimNow virtual machine needs a hard-drive image, which contains the Operative System to run. The GENIMAGE tool has the goal to create a customized version of a Linux distribution by other tools like VMBuilder and Debootstrap [137].

ii) ADD-IMAGE: this tool is preparatory to the GENIMAGE tool and it serves to load a given hard-drive image and the related virtual machine snapshot.

iii) BOOTSTRAP: it is a preparatory tool to prepare a user-based COTSon installation. This tool aims at solving the dependencies needed by the tool-set in the host machine, avoiding the manual installation of them. It requires root permission once per machine. Moreover, the tool tunes some kernel parameters such as the number of host memory pages needed by SimNow.

iv) CONFIGURE: it enables multiple users on the host machine to run a configuration of their own simulation setup without the need of system administrator inter-

vention. In fact, the tool runs completely in user-space, without the need of root permissions.

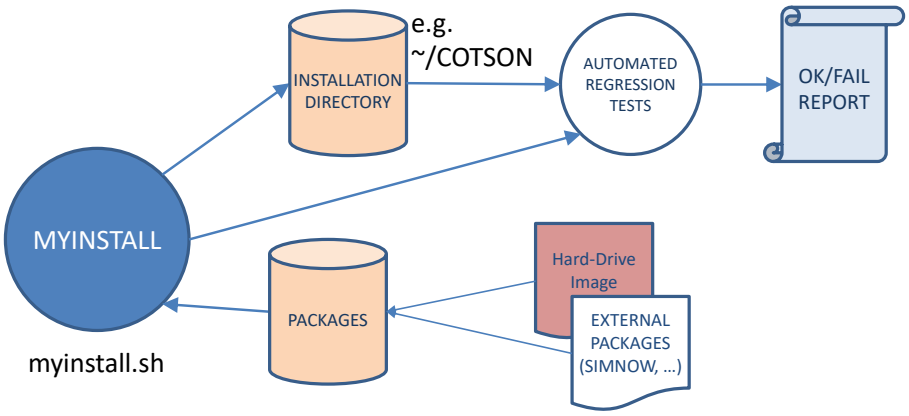

Figure 3.2: TOOLFLOW for the MYINSTALL tool. MYINSTALL prepares the whole environment for simulation-based Design Space Exploration with a single command. The Configuration File specifies which additional tool or tool-options should be used (e.g., non-public tools, or tools under NDA) [40].

v) MYINSTALL: the purpose of this tool is to facilitate the installation process of the simulator and the hard-disk image which contains the pre-selected Operative System that will runs into the SimNow machine (see Figure 3.2). Moreover, MYINSTALL allows the choice of the simulation software version, in order to enable more versions of the simulator to co-exist for regression test.
Finally, the tool performs several regression tests at the end of the installation phase, in order to verify the software is correctly patched, compiled and installed. The entire process is completely automatic and it can be easily repeated on multiple and parallel simulation hosts.

vi) MYDSE: we found a substantial need to implement a specific tool, which is able to easily catch possible failures or errors and, mostly, the automatic management of experiments in case of a large number of design points to be explored. As depicted in Figure 3.3, MYDSE relies on a small configuration file, named ***INFOFILE***, which is described in more details in the subsection 3.1.3.
Also, the tool is able to spread the simulation among multiple hosts and,if necessary, it can use the same binary with different GLIBC library version across the hosts. This allows us to use different operating systems, with a different version of the GLIBC library, in different guests. During the experiment, MYDSE controls the simulation loop, collecting in an ordered way the several files from each simulation point. Statistics based on user formulas are printed out at the end of each simulation, in order to provide an overall evaluation of the results. In case of failures, the tool kills the failed simulation and the related processes after a certain

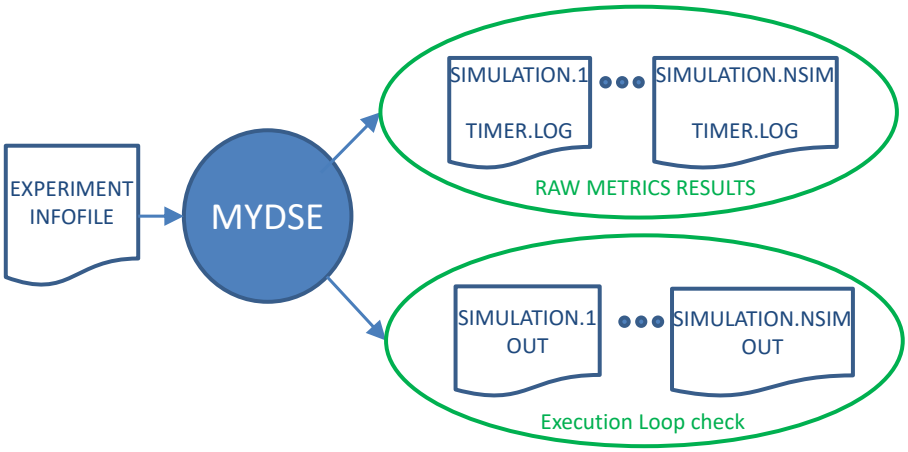

Figure 3.3: TOOLFLOW for the MYDSE tool. The experiment INFOFILE defines the simulation points and output files generated during each simulation are organized in order to facilitate their accessing and parsing by the other tools [40].

time, trying the re-execution of the failed simulation automatically. The timeout is derived by a simulation estimation model (i.e., proportional to the number of nodes and cores of the system).

vii) GTCOLLECT: once a campaign of experiments has been concluded, we need to collect, analyze and plot results in a simple way. In this perspective, we can extract data from experiments with the GTCOLLECT tool (GT stand for Graphic Table Collect), which prints out the collected data, based on the ***INFOFILE*** information and a ***LAYOUT*** text files where the user can specify the relevant output metrics to select. Furthermore, some additional calculations are performed on the data, such as the Coefficient of Variation, in order to analyze the correctness of the results. With the GTCOLLECT tool, we can perform a complete analysis of the raw data produced by the MYDSE.

viii) GTGRAPH: once the results are collected in the GTCOLLECT format, the GTGRAPH tool can produce a graphical view of the data, like Figure 7,8,9.

Additionally, COTSon permits a connection with McPAT [41] to analyze the power consumption and the temperature of an experiment.

3.1.3 Experiment description

In this section, we want to introduce the experiment description file, named ***INFOFILE***, which makes the DSE easy to manage a clear identification of the Design Space. As depicted in Figure 3.4, we can describe the experiment through a simple file that uses "Bash syntax": <variable> = "<string>". Each DSE variable is defined with the prefix "list", while "<string>" represents a set of value where elements can be separated with

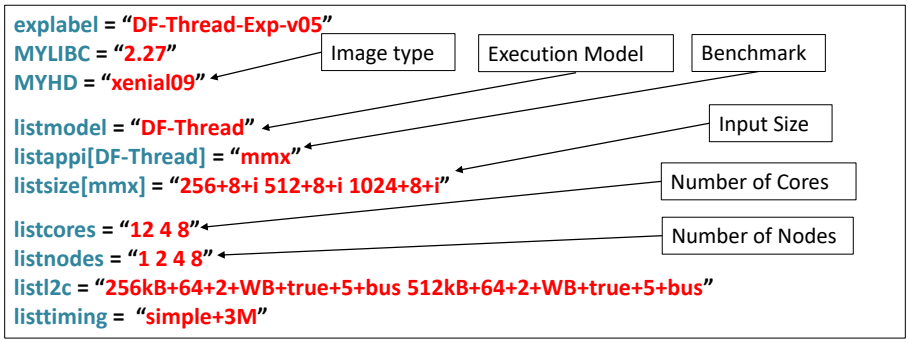

Figure 3.4: INFOFILE example, which describes a Design Space Exploration experiment [40].

the character "+" (i.e. 256+8+i represent matrix size = 256, block size = 8 and matrix element type = integer). Moreover, we can define multiple configurations of the architecture, in order to find the best organization for a given application.

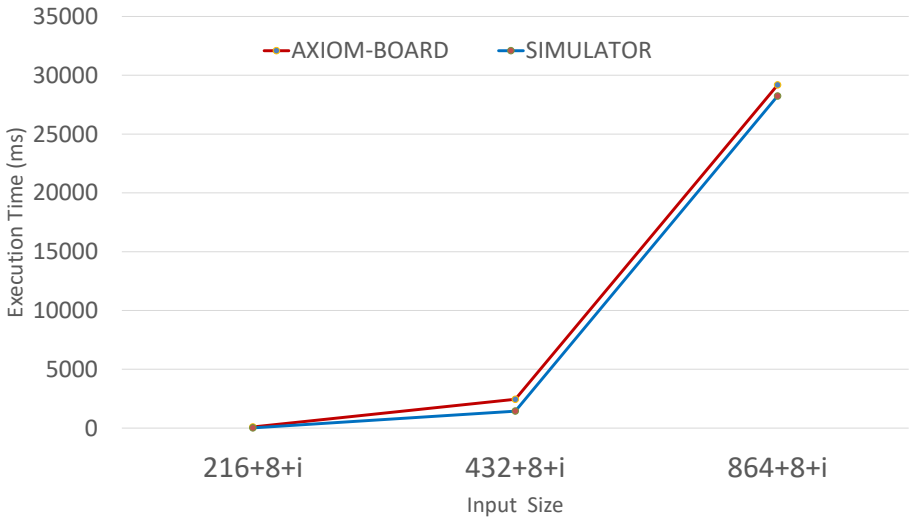

Figure 3.5: Preliminary comparison of the execution time between the simulator and the AXIOM-Board. We used the Matrix Multiply benchmark three different size: 216,432 and 864.

3.1.4 Performance validation

Finally, we validated our results obtained through the utilization of the simulator, comparing the execution time (sequential execution) of the Matrix Multiplication Benchmark both in simulator and on a real board FPGA based (AXIOM-Board [106], [39], [32], [2]). As can be seen in Figure 3.5, the results of the simulator and the board are very close, confirming our performance predictions, despite some architectural differences.

3.2 Operating system impact

In this section, we present the tests performed into a distributed simulation environment in order to study the influence of several Linux distribution on the performance. For the comparison, we selected the DF-Threads execution model and the Matrix Multiplication Benchmark. We analyzed how the execution time differs varying the Operating System. Moreover, we studied key aspects like ***Kernel Cycles*** to evaluate the impact of each distribution on the execution time.

We configured the distributed simulation environment with a node range from 1 to 4 and a core range from 1 to 32. The input sizes used for the Matrix Multiplication benchmark vary from 64 to 1024.

The operating system are based on the Linux distribution Ubuntu like and we chose four different kernel versions to perform our tests:

- Karmic: it is the Ubuntu 9.10 LTS version. The distribution focuses on improvements in cloud computing as well as further improvements in boot speed.
- Tfx: it represents the Maverick version of the Ubuntu kernel (version 10.10)
- Trusty (Ubuntu 14.04 LTS): the main improvements were based on increasing the performance and the maintainability.
- Xenial: It is the Ubuntu 16.04 LTS version

3.2.1 Performance variations (Cycles)

The first comparison between the Linux distribution regards the execution time and the scalability. We vary the number of nodes (from 1 to 4), keeping fixed the input size (matrix size=512) and the cores number. As we can see in Figure 3.6, the Trusty distribution obtains better results, both in execution time and scalability, than the other releases, outperforming the Xenial version by a factor of 60% in the one node execution. This variation rises from the different configurations and daemons included in the distro. Moreover, the Karmic distribution showed a quite similar result as the Trusty when we increase the number of nodes, confirming the improvements in cloud computing and performances made by the Linux development communities. Once we increase the number of nodes, one interesting effects is also that the total capacity of caches will increase. Moreover, we observe a reasonable scaling of performance with the number of nodes.

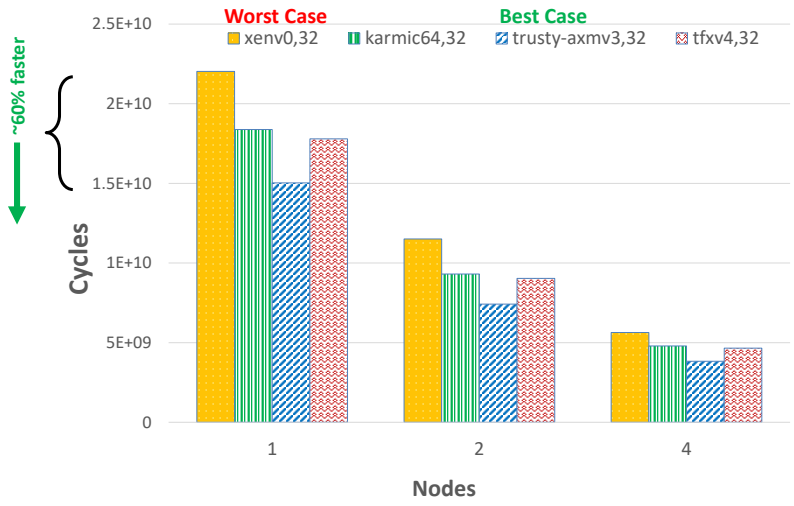

Figure 3.6: Study of the performance variation (cycles) as we vary the number of nodes with different operating system distribution. In the case of one node, this variation is up to 60% due to the different types of daemons and configurations included in distros.

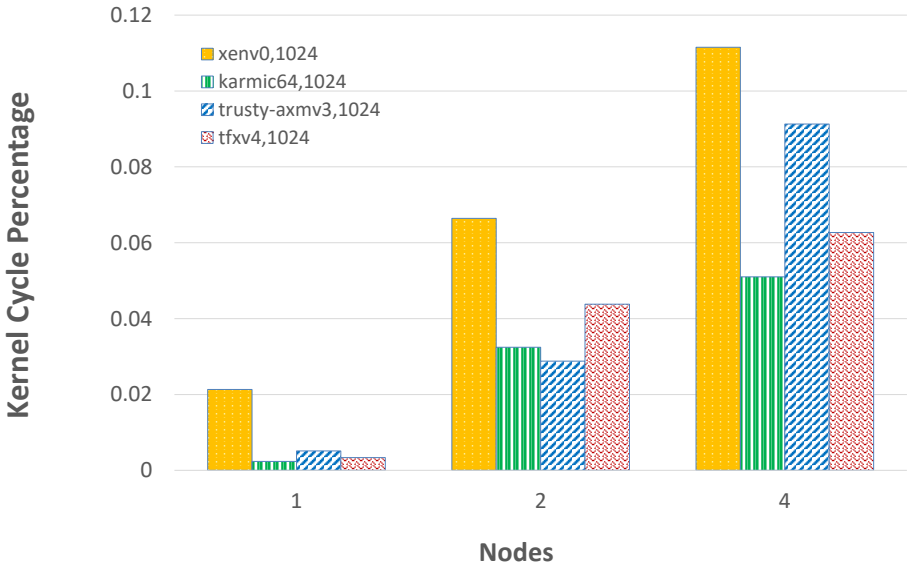

Figure 3.7: The percentage of kernel cycles for different types of OS while increasing number of nodes. The kernel time may occupy from few percent (1 Node execution) to more than 13% of the total time (4 Nodes execution based on the DF-Threads execution model).

3.2.2 Kernel activity

Regards to study the influence of the kernel execution time on the total cycles of the application, we configured the experiment keeping fixed the L2 cache (256 KiB), varying the matrix input size (from 64 to 1024), the number of nodes (from 1 to 4) and the OS distributions. As we can observe in Figure 3.7, the Karmic distribution shows better results, keeping the percentage of the kernel cycles lower than the others releases in any configuration. Generally, the kernel impact on the total cycles is proportional to

the number of nodes, implying that the influence of the kernel is more important for multi-node configurations.

3.3 Final remarks

We have presented a set of tools for the Design Space Exploration, based on the COTSon simulator framework, with the aim of supporting large set of experiments of a multi-node multi-core platform with full OS execution (e.g., 1,2, and 4 nodes and real OS activity). Thanks to our tools, we setup the simulation environment in less than ten minutes, including several regression tests, saving hours in comparison with manual installation. We can run several experiments by using a simple configuration file, handle possible failures or errors during the simulations. Finally, results are automatically collected and presented in the desired graphical view.
We showed a validation test against a FPGA platform. The results permitted us to derive information early in the design process.
The DSE tools that we presented in this paper were massively used during two European projects (TERAFLUX and AXIOM), facilitating the exploration of a large design space and the test of a new execution model (DF-Threads).

Chapter 4

Translating Timing Model into High-Level Synthesis (HLS)

4.1 Introduction

In recent decades, applications are becoming more and more sophisticated and that trend may continue in the future [138]–[140]. To cope with the consequent system design complexity and offer better performance, the design community has moved towards design tools that are more powerful. Today many designs rely on FPGAs [141], [142], in order to achieve higher throughput and better energy efficiency, since they offer spatial parallelism on the portion of application characterized by dataflow concurrent execution. FPGAs are becoming more capable to integrate quite large designs and can implement digital algorithms or other architectures such as soft-processors or specific accelerators [142]. For the efficient use of FPGAs, it is essential to have an appropriate tool-chain. The tool-chain provides an environment, in which the user can define, optimize and modify the components of the design, by taking into account the power, performance, and cost requirements of a particular system and eventually synthesize and configure the FPGA.

The conventional method to implement application code on FPGAs is to write the code in Hardware Description Language (HDL) (e.g., VHDL or Verilog). Although working with HDL languages still is the most reliable and detailed way of designing the underlying hardware for accelerators, their use requires advanced expertise in hardware design as well as remarkable time. The Design Space Exploration (DSE) and debugging time of FPGAs and the bitstream generation may reach many hours or days even with powerful workstations. As such, moving an already-validated architecture to the FPGA's tool-flow may save significant time and effort and, as a result, facilitates the design development.

This situation is exacerbated by the interaction with the Operating Systems and by the presence of multi-core (see Section 3.2). Therefore, the use of full-system simulators in combination with HLS tools permits a more structured design flow. In such case, a simulator can preliminary validate an architecture and the HLS-to-RTL time is repeated less times.

Based on the experience of the previous projects like TERAFLUX [81], [51], [52], ERA [143], [144], AXIOM [141], [36], [37], SARC [145], we choose to rely on the HP-Labs COTSon simulation infrastructure [41]. The key feature of COTSon that is useful in HLS design is its "functional-directed" approach, which separates the functional simulation from the timing one. We can define custom timing models for any component of an architecture (e.g., FPGA, CPU, Caches) and validate them through the functional execution: however, the actual architecture has to be specified by a separate "timing model" [57]. The latter is what can be migrated in a straightforward way to HLS.

Moreover, COTSon is a full-system simulator, hence it permits to study the OS impact on the execution and choose the best OS configuration based on the application requirements [146]; the OS modeling is sometimes not available in other tools (discussed in section 4.2). In this Chapter, we illustrate the importance of the simulation in synergistic combination with the Xilinx HLS tool, in order to permit a faster design environment, while providing a full-system Design Space Exploration (DSE). Additionally, thanks to our DSE tool-set (MYDSE) [40], [146], we facilitate the extraction of important metrics, which help investigate the appropriate system design. In order to illustrate our methodology, we start from a driving example related to design a simple two-way associative cache system. The methodology is then generalized by considering the case of the AXIOM project, in which this methodology was actually used to design and implement a novel Data-Flow architecture [51], [147], [106] through the development of our custom AXIOM-Board [141].

This chapter presents our methodology for designing FPGA-based architectures, which consists in the direct mapping of COTSon "timing models" into the HLS, where such models are pre-verified via our MYDSE tools: the DSE facilitates prototyping, timing model analysis, and help save much design time. The DSE tool-set is designed and developed at our research group. We illustrate a simple driving example based on the modeling and synthesis of a simple two-way set-associative cache in order to grasp the details of our methodology. The proposed methodology is used to develope a the AXIOM platform (funded by the Horizon 2020 - european commission).

Candidate's contribution on this chapter are:

1. Mapping the "timing models" into the HLS.

2. Validating the AXIOM board against the COTSon simulator by running BMM (Blocked Matrix Multiplication).

4.2 High-Level Synthesis (HLS) tools

Our design and evaluation methodology aims at integrating simulation tools and HLS tools to ease the hardware acceleration of applications, via custom programmable logic. HLS tools improve design productivity as they may provide a high level of abstraction for developing High-Performance Computing systems. Most typically, these tools allow users to generate a RTL representation of a specific algorithm usually written in C/C++ or SystemC. Several options and features are included in these tools in order to provide an environment with a set of directives and optimizations that help the designer meet the overall requirements. In our case, we realized that more design productivity could be achieved by identifying in the early stages a candidate architecture through the use of a simulator. However, the use of a generic simulator may not help identify the architecture, since often the simulation model is too distant from the actual architecture or is too much intertwined with the modeling tool [148]–[151]. On the other hand, the COTSon simulator is using a different approach, called “functional directed” simulation, in which the functional and timing models are neatly separated and the first one drives the latter. The similarity of our “timing model” specification to an actual architecture is an important feature and it is the basis for our mapping to a HLS specification.

In our research, we used Xilinx Vivado HLS, but other important HLS frameworks are available and are briefly illustrated in the following; their main features are summarized in Table 4.1 (The format and content of the table is adopted from the PhD thesis of Dr. Emanuel del Sozzo [152]). LegUp [153] supports C/C++, Pthreads and OpenMP as programming models for HLS by leveraging the LLVM compiler framework [154], and permits parallel software threads to run onto parallel hardware units. LegUp can generate customized heterogeneous architectures based on the MIPS soft processor. Bambu [155] is a modular open-source HLS tool, which aims at the design of complex heterogeneous platforms with a focus on several trade-offs (like latency versus resource utilization) as well as partitioning on either hardware or software. GAUT [156] is devoted to real-time Digital Signal Processing (DSP) applications. It uses SystemC for automatic generation of test-benches for more convenient prototyping and design space exploration.

Table 4.1: Key features of discussed HLS tools. For the non-obvious columns, Testbench means the capability of automatic testbench generation. SW/HW means the support for the Software/Hardware co-design environment. Floating Point and Fixed Point are the supported data types for the arithmetic operations (The format of table is adopted from [152]).

Tool	*Owner*	*License*	*Input*	*Output*	*Domain*	*Testbench*	*SW/HW*	*Simulation*	*Floating Point*	*Fixed Point*
LegUp	LegUp Computing	Commercial	C, C++	Verilog	All	Yes	Yes	HW	Yes	No
Bambu	Politecnico di Milano	Academic	C	VHDL, Verilog	All	Yes	Yes	SW, HW	Yes	No
GAUT	U. Bretagne Sud	Academic	C, C++	VHDL, SystemC	DSP	Yes	No	HW	No	Yes
DWARV	TU Delft	Academic	C	VHDL	All	Yes	Yes	HW	Yes	Yes
Stratus HLS	Cadence	Commercial	C, C++, SystemC	C, C++, SystemC	All	Yes	Yes	SW, HW	Yes	Yes
Intel HLS Compiler	Intel	Commercial	C, C++,	Verilog	All	No	No	SW, HW	Yes	Yes
Vivado HLS	Xilinx	Commercial	C, C++, OpenCL, SystemC	VHDL, Verilog, SystemC	All	Yes	No	SW, HW	Yes	Yes
SDSoC	Xilinx	Commercial	C, C++	VHDL, Verilog	All	No	Yes	SW, HW	Yes	Yes
SDAccel	Xilinx	Commercial	C, C++, OpenCL	VHDL, Verilog, SystemVerilog	All	Yes	Yes	SW, HW	Yes	Yes

DWARV [157] supports a wide range of applications like DSP, multimedia, encryption. The compiler used in DWARV is the CoSy commercial infrastructure [158], which provides a robust and modular foundation extensible to new optimization directives. Stratus HLS of Cadence [159] is a powerful commercial tool accepting C/C++ and SystemC and targeting a variety of platforms including FPGAs, ASICs, and SoCs. Thanks to low power optimization directives, the user can achieve a consistent power reduction. It gives support for both control flow and data flow designs, and actively applies constraints to trade-off speed, area, and power consumption. Intel HLS Compiler [160] accepts ANSI C/C++ and generates RTL for Intel FPGAs, which is integrated into the Intel Quartus Prime Design Software. Xilinx Vivado HLS tool targets Xilinx FPGAs, which offers a subset of optimization techniques including loop unrolling, pipelining, dataflow, data packing, function inline, bit-width reduction for improving the performance and the resource utilization.

Xilinx SDSoC is a comprehensive automated development environment for accelerating embedded applications [36]. The tool can generate both RTL level and the software running on SoC cores for the ***bare-metal*** libraries, Linux, FreeRTOS. Xilinx SDAccel [161] aims at accelerating functionalities in data-centers through FPGA resources. We summarize the key features of aforementioned HLS tools in Table 4.1.

4.3 Simulator tools

Although some of the HLS tools provide a general Software/Hardware simulation framework, the possibility of easily evaluating a complex architecture oriented design (e.g., computer organization: level and size of caches, number of cores/nodes, memory hierarchy) is still missing. Moreover, before reaching a bug free physical design, which meets all the design specifications, the debug and development of such designs by using aforementioned HLS tools may require a significant time and effort despite all benefits that HLS tools provide to the design community. Consequently, powerful design frameworks that simplify the verification of the design and provide an easy design space exploration are welcome. In this respect, many design frameworks have emerged to implement efficient hardware in less time and effort. The authors in [162] propose a framework relying on Vivado HLS to efficiently map processing specifications expressed in PolyMageDSL to FPGA. Their framework support optimizations for the memory throughput and parallelization. ReHLS [163] is a framework with automated source-to-source resource-aware transformation leveraging Vivado HLS tool. Their framework improves the resource utilization and throughput by identifying the program inherent regularities that are invisible by HLS tool. FROST [164] is a framework that generates an optimized design for HLS tool. This framework is mainly appropriate for applications based on streaming dataflow architectures like image processing kernels.

However, these tools focus on optimizing the whole application performance, while we are proposing instead an architecture oriented approach, where the designer can manipulate and explore the architecture itself, before passing it to the HLS tool-chain. By using our proposed framework (see Section 4 for more details), we can validate the design in terms of the functional and timing models, and then define a specific architecture, while constantly monitoring the selected key performance metrics. The architecture model is specified in C/C++ and, thanks to the decoupling from the simulation details and functional model, it can be easily migrated into the HLS description. This is illustrated in Sections 4.5 and 4.6. In particular, we leverage the Vivado HLS tool and on top of it, we build our design space exploration tools relying on COTSon simulator, which is one of the key components of our framework. In the following, we highlight relevant features and compare several simulators (Table 4.2), and we contrast them with our chosen simulator (i.e., COTSon).

SlackSim [148] is a parallel simulator to model single-core processors. Simple-Scalar [149] is a sequential simulator, which supports single-core architectures at user-level.

GEMS [150] is a virtual-machine based full-system multi-core simulator built on top of the Intel's Simics virtual-machine. GEMS relies on timing-first simulation approach, where its timing model drives one single instruction at a time. Even though GEMS provides a complete simulation environment, we found that COTSon simulator provides better performance as we increase the number of modeled cores and nodes. MPTLsim [151] is a full-system x86-64 multi-core cycle-accurate simulator. In terms of simulation rate, MPTLsim is significantly faster than GEMS. MPTLsim takes advantage of a real-time hypervisor scheduling technique [164] to build hardware abstractions and fast-forward execution. However, during the execution of hypervisor, the simulator components like memory, instructions or I/O are opaque to the user (no statistics is available). On contrary, for example, COTSon provides an easily configurable and extensible environment to the users [165] with full detailed statistics. Graphite [166] is an open-source distributed parallel simulator leveraged the PIN package [44], with the trace-driven functionalities. COTSon permits full-system simulation from multi-core to multi-node and the capability of network simulation, which makes COTSon a complete simulation environment. Both COTSon and Graphite permit a large core numbers (e.g., 1000 cores) with reasonable speed, but COTSon provides also the modeling of peripherals like disk and Ethernet card as well. Compared to COTSon, the above simulators do not express a timing model in a way that can be easily ported to HLS: COTSon is based on the "functional directed" simulation [41], which means that the functional part drives the timing part and the two parts are completely separated both in the coding and during the simulation. The functional model is very fast but does not include any architectural detail, whilst the timing model is an architectural-complete description of the system (and, as such, includes also the actual functional behavior, of course). In this way, once the timing model is defined and the desired level of the key performance metric (e.g., power or performance) has been reached, the design can be easily transported to an HLS description as it will be illustrated in the next Sections.

There are other light-weight simulators targeting RISC-V ISA like WebRISC-V [167], which is developed by our research group. WebRISC-V is a web-based server-side RISC-V assembly language Pipelined Datapath simulation environment, which aims at easing students learning and instructors teaching experience.

4.4 Methodology

In this section, we present our methodology (Figure 1) for developing hardware components for a reconfigurable platform, as developed in the context of the AXIOM project.

First, we define the functional and the timing model of a desired architectural component (e.g., a cache system, as described in the section 4.5). Such models are described

Table 4.2: Interesting features of simulators for high performance computing architectures. For the non-obvious columns, Parallel/Sequential means the simulator core can be executed either in parallel or sequential by the host processor. Full System means taking into account all events, including the OS.

Simulator	***Parallel/ Sequential***	***Single-core/ Multi-core***	***Full System***	***Simulation methodology***
COTSon	***Parallel***	***Multi-core***	***yes***	***Decoupled- functional first***
GEMS	Sequential	Multi-core	yes	Decoupled – timing first
Graphite	Parallel	Multi-core	no	Not-Decoupled – trace driven
SimpleScalar	Sequential	Single-core	no	Not-decoupled – execution driven
MPTLsim	Sequential	Multi-core	no	Not-decoupled – timing first
SlackSim	Parallel	Single-core	no	Not- decoupled – timing first

by using C/C++ (two orange blocks in the top left part of Figure 4.1). These models are then embedded in the COTSon simulator, which is managed in turn by the MYDSE tools in order to perform the design space exploration [146],[41]. The latter are a collection of different tools, which provide a fast and convenient environment to simulate, debug, optimize and analyze the functional and timing model of a specific architecture and to select the candidate design to be migrated to the HLS (top part of Figure 4.1). The detailed discussion regarding the MYDSE tools and COTSon simulator is carried out in Section 3.

Afterwards, we manually migrate a validated architecture specification from COTSon to Vivado HLS tool (bottom part of Figure 1), where the user can apply the specific directives defined in the timing model of COTSon into the Vivado HLS. This is possible due to the close syntax of the architecture specification in COTSon and Vivado HLS. Our framework has the purpose of reducing the total DSE time to define an architecture (as input to Vivado HLS itself). We do not aim to define a precise RTL, but simply to select an architecture suitable as input to Vivado HLS (see 4.2).

Finally, we pass the generated bitstream by Vivado to the XGENIMAGE, which is a tool that assembles all needed software including drivers, applications, libraries and packages in order to generate the operating system full image to be booted on the AXIOM board. In Figure 4.1, we highlight in green the existing (untouched) tools, in blue we highlight the research tools that we developed from scratch or that we modified (like COTSon). In our case, part of the process involves the design of the FPGA board (the AXIOM board). An important capability of the board is also to provide fast and

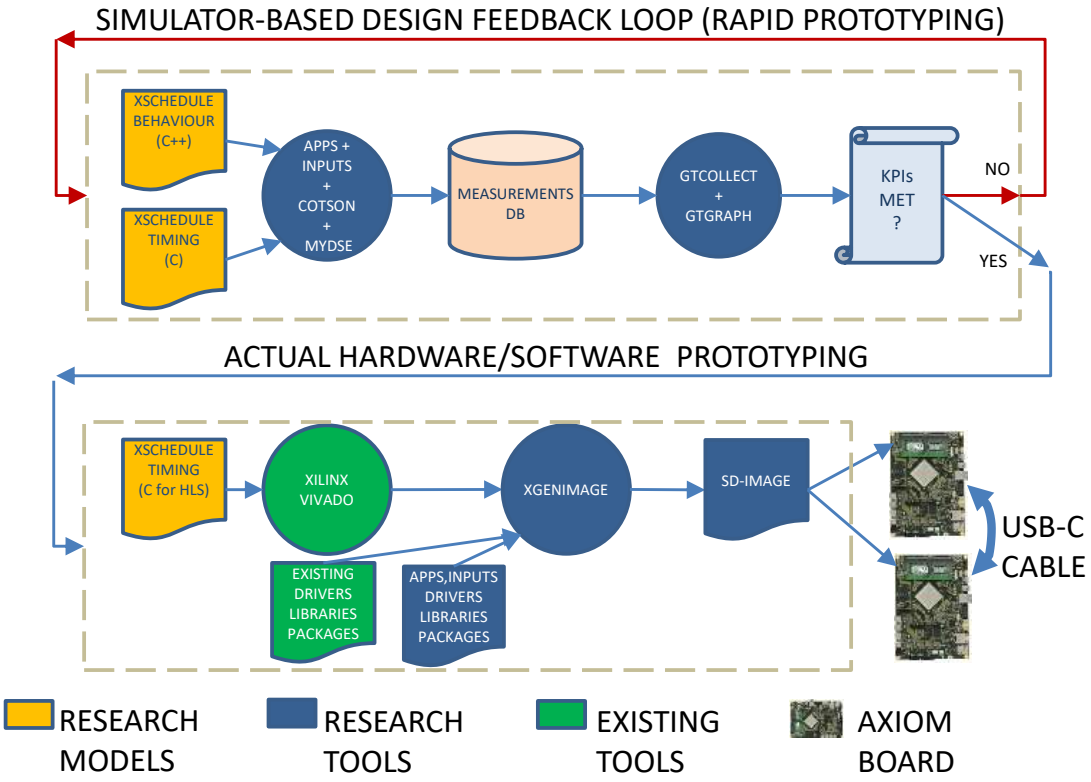

Figure 4.1: The design and test methodology of the AXIOM involved a mix of simulation (via the COTSon simulator and other custom tools) and FPGA-prototyping (via our custom AXIOM board and hardware synthesis tools (like Vivado HLS) [39].

inexpensive clusterization. The simulator allowed us to model exactly this situation, in which the threads are distributed across several boards, through a specific execution model (called DF-Threads). To that extent, the AXIOM board [39] has been designed to include a soft-IP for the routing of data (via RDMA custom messages) and the FPGA transceivers are directly connected to USB-C receptacles, so that four channels at about 18Gbps are available for simple and inexpensive connection of up to 255 boards, without the need of an external switch [37].

4.4.1 Mapping Architecture to HLS

High-Level Synthesis (HLS) aims at enhancing design productivity via facilitating the translation from the algorithmic level to RTL (Register Transfer Level) [168], [169]. In current state-of-the-art, given an application written in a language like C/C++ or SystemC, an HLS tool particularly performs a set of successive tasks to generate the corresponding Register Transfer Level (RTL, for example, VHDL or Verilog) description suitable for a reconfigurable platform, such as an FPGA [169] (Figure 4.2 - left). This workflow typically involves the following steps:

1. Compiling the C/C++/SystemC code to formal models, which are intermediate representations based on control flow graph and data flow graph.

2. Scheduling each operation in the generated graph to the appropriate clock-cycles. Operations without data dependencies could be performed in parallel, if there are enough hardware resources during the desired cycle.

3. Allocating available resources (LUTs, BRAMS, FFs, DSPs and so on) in regards to the design constraints. For instance to enhance the parallelism, different resources could be statistically allocated at the same cycle without any resource contention.

4. Binding each operation to the corresponding functional units, binding the variables and constants to the available storage units as well as data paths to data buses.

5. Generating the RTL (i.e., VHDL or Verilog).

All these operations continue to be performed in our proposed framework, but the designer would like to avoid excessive iterations through them, since they may require many hours of computing processing or even more, depending on the complexity of the design, even on powerful workstations and with not so big designs. However, COTSon and MYDSE tools (illustrated above) act like a "front-end" to the HLS tool, as outlined in Figure 4.2. We use HLS also for defining a specific architecture to accelerate the application. Our tools allow the designer to explore possible options for the architecture, without going to the synthesis step: only when the simulation phase has successfully selected an architecture (output of the blue block in Figure 4.2), the model will be manually translated by the programmer as an input to the HLS synthesis tools. Doing this step automatically is out of the scope of this work.

A comparison of the total time of the DSE loops between our framework (Figure 4.2 - right) and HLS (Figure 4.2 - left) is reported here for different benchmarks (Table 4.3). For example, a Blocked Matrix Multiplication benchmark (matrix size 864, and block size 8), and a Fibonacci benchmark (order of up to 35) are executed based on our DF-Thread execution model (Data-Flow model). As a result, thanks to our framework, we were able to reduce the required time in validating and developing the architecture compared with solely HLS workflow, through which applying any changes in the source codes may require many hours for the synthesis process.

4.5 Case study

In this section, first we want to explain our workflow by using a simple and well-known driving example, i.e., the design of a two-way set associative cache in a reconfigurable

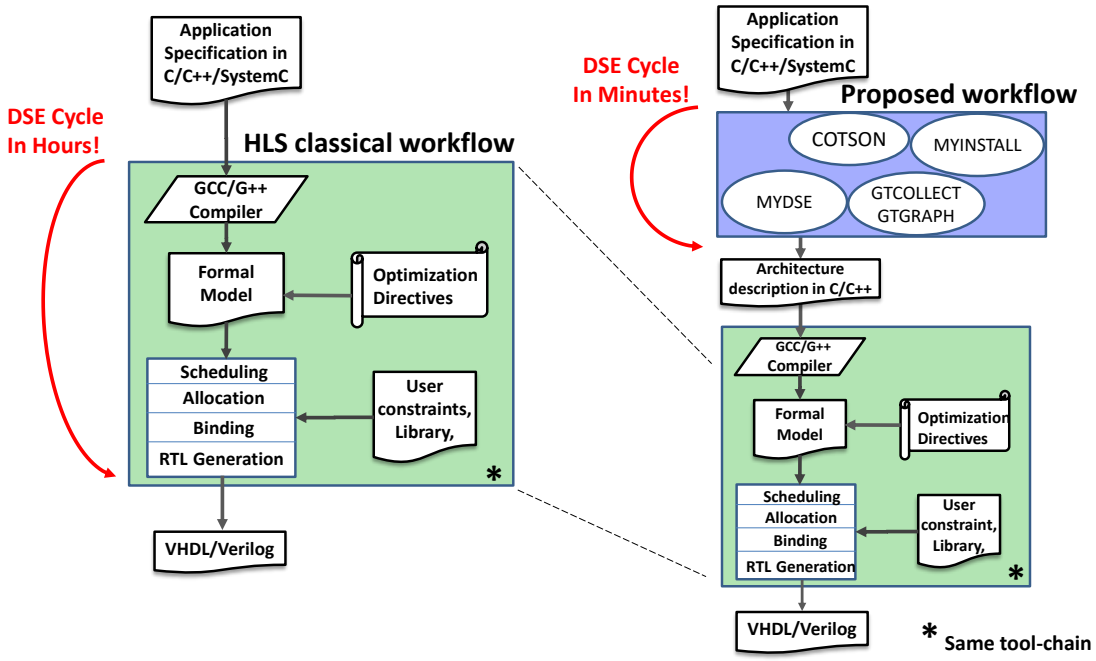

Figure 4.2: Differences between classical and proposed architecture modelling framework. Workflows to generate VHDL/Verilog hardware description language from the application specification written in C/C++. On the left, a typical workflow of existing HLS tools. On the right, we leverage the HLS tool, and on top of it, we build our framework to simulate and validate the design specification [57].

Table 4.3: Comparison of different total DSE time of the classical design workflow for FPGAs (Figure 4.2 - left) and our proposed methodology (Figure 4.2 - right).

Application	*HLS+Synthesis*	*Our Framework*
2-Way Cache	3:50 Hours	5 Seconds
Blocked Matrix Multiplication (DF-Threads, matrix size = 864, block size=8, integer)	4:25 Hours	8 Seconds
Fibonacci (DF-threads, N=35)	1:40 Hours	8 Seconds

hardware platform through our methodology. Afterwards, we will illustrate the more powerful capabilities of our framework for a more complex example, which is the design of the AXIOM hardware/software platform. In both cases, first we design the architecture in the COTSon simulator, then we test its correct functioning and achieve the desired design goals. Finally, we migrate the timing description of the desired architecture into the Xilinx HLS tools.

4.5.1 From COTSon to Vivado HLS – a simple example

In COTSon, the architecture is defined by detailing its "timing model". A timing model is a formal specification that defines a custom behavior of a specific architectural or micro-architectural component, in other terms the timing model defines the architecture itself [41], [40]. The timing model in the COTSon simulator is specified by using C/C++. The designer defines the storage by using C/C++ variables (more often structured variables). The timing model behavior is specified by explicating into C/C++ statements the steps performed by the control part and associating them with the estimated latency, which can be defined through our DSE configuration files (in a lua file formate [57]) easily. After defining the model, we can simulate and measure the performance of it. This is illustrated in Figure 4.3, and discussed in the following.

Let us assume here that we wish to design a simple two-way set-associative cache: we show how it is possible to define the timing model of a simple implementation of it in COTSon and then how we can map it in HLS. We start here from a conceptual description of such cache, as shown in Figure 4.3. In particular, for each way of the cache, we need to store the "line" of the cache, i.e., the following information:

1. Valid bit or V-bit (1 bit): used to check the validity of the indexed data;
2. Modify bit or M-bit (1 bit): used to track if data has been modified;
3. LRU bits or U-bits (e.g., 1 bit in this case): used to identify the Least Recently Used data between the two cache ways;
4. Tag (e.g., 25 bits): used to validate the selected data of the cache;
5. Data (e.g., 512 bits, 64 bytes, or 16 words): contains the (useful) data.

The data structures to store this information in COTSon is given by the "Line" structure, which is shown in Figure 4.4 (left side).

When we want to read or write data, which are stored in a byte address (X in Figure 4.3), we check if the data are already present into the cache. The cache controller implements the algorithm to find the data in the cache. Although not visible in the left part of Figure 4.3, there is a control part also for identifying the LRU block. We can implement this control in COTSon by using the two functions (shown in the right part): one named "find" (Figure 4.4), which is a simple linear search, and the other one named "find_lru" (Figure 4.5).

4.6 Generalization to the AXIOM project

The aim of the AXIOM project was to define a software/hardware architecture configuration, to build scalable embedded systems, which could allow a distributed computa-

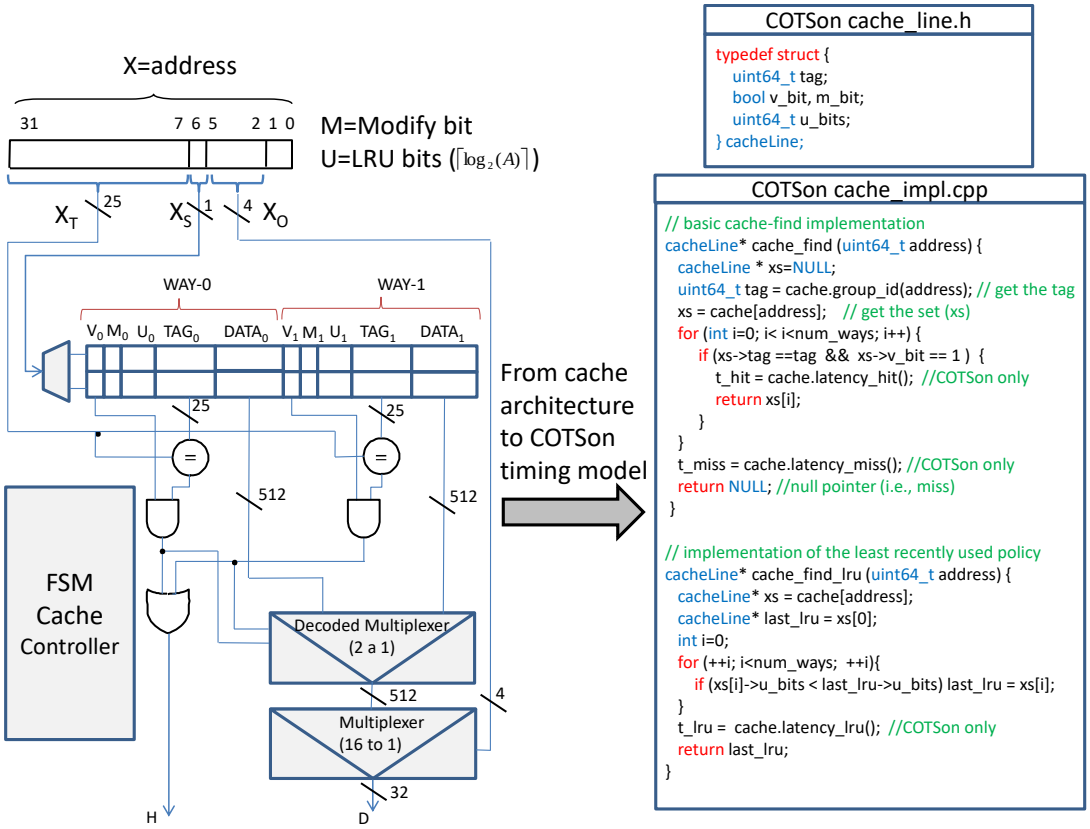

Figure 4.3: Example of logic scheme of a two-way set associative cache. Given the byte address X on 32 bits, in this example, the cache indexes four 64-byte blocks (2 words in 2 sets). This implies that the last 6 bits are needed to select a byte inside the block, the first 25 bits of the address (XT) is used for the tag comparison and the remaining 1 bit (Xs) is used for the cache set indexing. The cache hit (signal H) is set if the tag of the X is present in the cache at the specified index and if the valid bit is equal to one [57].

tion across several boards by using a transparent scalable method like the DF-Threads [1], [147], [106].

An essential contribution of the AXIOM was the fabrication of an SBC board (***AXIOM board***) based on FPGA and embedded processor, e.g. Zynq Ultrascale+ [170]. and its features are: i) a high speed reconfigurable interconnect for board-to-board communication; and ii) a user-friendly programmable environment, which allows us both to off-load partly program algorithms into accelerators (on programmable logic) and, at the same time, to distribute the computation workloads across boards via Data-Flow-Threads, a novel execution model [1], [50], [106] and iii) the possibility of deploying an open-source tool-chain based upon easy to program concept like OmpSs [171], an OpenMP extension [172], [173].

In order to achieve this goal, we rely on Remote Direct Memory Access (RDMA) capabilities [174] and a full operating system to interact with the OS scheduler, memory

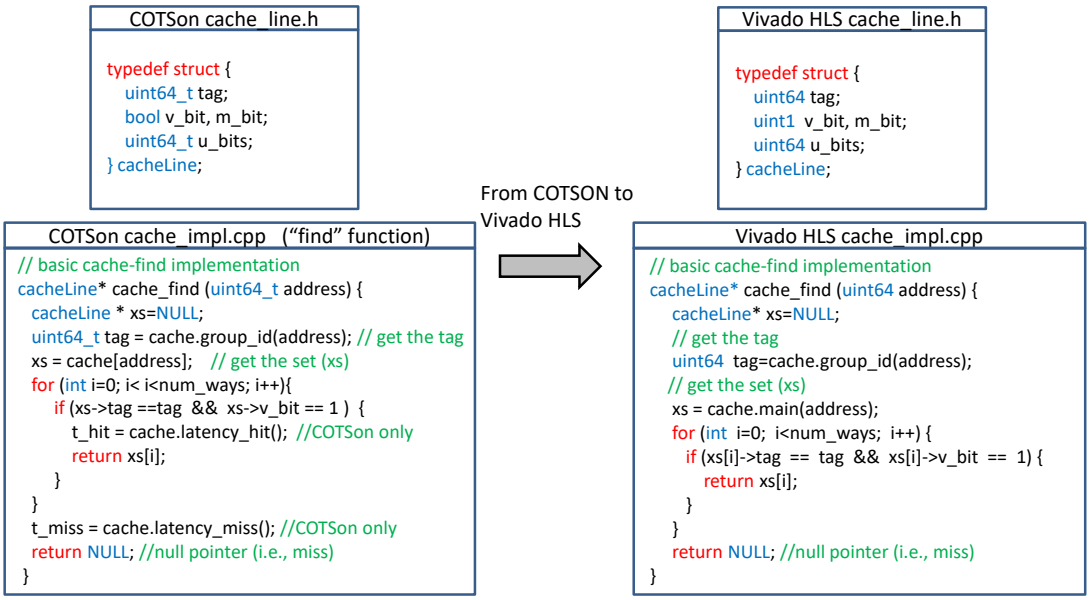

Figure 4.4: Example of timing model of the cache find function, which is translated from the COTSon to the Vivado HLS. The implementation of this function for the both COTSon (left) and Vivado HLS (right) environments is shown in the bottom part of the figure [57].

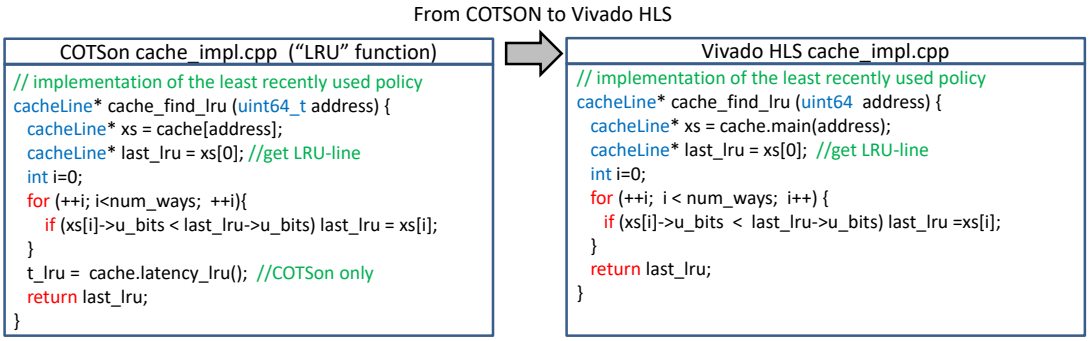

Figure 4.5: Example of translation of the timing model of the LRU (Least Recently Used) function from the COTSon (left side) to the Vivado HLS (right side) [57].

management and other system resources. Following our methodology, we included the effects of all these features thanks to the COTSon+MYDSE full-system simulation framework. We will present in the next subsection the results that we were able to obtain through this preliminary DSE phase reasonably quickly.

After the desired software and hardware architecture was selected in the simulation framework, we started the migration to the physical hardware: we had clear that we needed at least the following features:

1. Possibility to exchange rapidly data frames via RDMA across several boards: this

could be implemented in hardware thanks to the FPGA high-speed transceivers;

2. Possibility to accelerate portions of the application on the Programmable Logic (PL), not only on one board but also on multiple FPGA boards: this could be implemented by providing appropriate network-interface IPs in the FPGA.

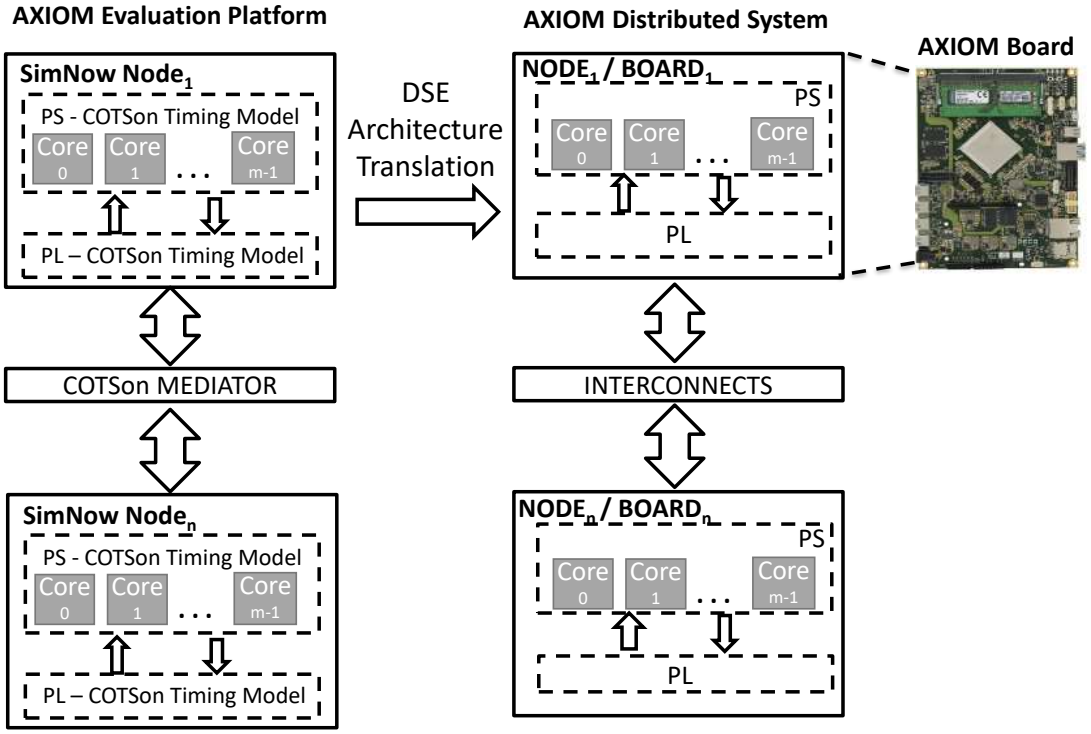

Figure 4.6: From the COTSon Distributed System definition to the AXIOM Distributed System by using the DSE Tools. The Processing System (PS), the Programmable Logic (PL) and the Interconnects of the AXIOM Board are simulated and evaluated into the COTSon framework with the definition of the respective timing models [57].

In this way, we preselected the basic features of the AXIOM board (Figure 4.6 - left) through the COTSon framework and the MYDSE tool-set (addressed in Section 3.1.2. Then, once the DSE was completed, we migrated the final architecture specification with the Vivado HLS tool into the AXIOM Distributed Environment (Figure 4.6 - right).

The DF-Threads execution model is a promising approach for achieving the full parallelism offered by a multi-core and multi-node systems, by introducing a new execution model, which internally represents an application as a direct graph named Data-Flow graph. Each node of the graph is an execution block of the application and a block can execute only when its inputs are available [1].

4.6.1 The AXIOM board

With the fast improvements in science, technology, and engineering, designers are gradually redefining the capabilities of computing systems around us to improve the so called "Embedded Intelligence" or "Smart Things". In fact, both "things" and people are becoming nodes of the same network, creating a Cyber Physical domain [175]. CPSs operate through intelligent interfaces to communicate via web and social media and interact with the environment. In our daily life, CPSs devices provide us with efficiency, flexibility such as in the case of smartphones, smart home and assisted/autonomous driving. Therefore, the growing number of applications have created a huge fragmentation in hardware platforms and software tools. Also, some of the main challenges in designing a CPS architecture are data management, proper software-hardware integration, real-time management and hardware specialization. By building upon successful examples of design-for-simplicity, like in the case of Arduino[176] and UDOO[177], the AXIOM platform (Agile, eXtensible, fast I/O Module) [32], [33], [35], [34], [2], [36], [37], [178] aims to provide a complete and general software development suite for easily mapping applications into multi-board processing systems.

According to known road maps for future systems, the crucial problems for a broader deployment of scalable embedded systems are easy programmability, and inexpensive ways to build a system based on the simpler components. The AXIOM project has defined a simple but powerful architecture that can possibly be deployed in CPS, since it includes not only the conventional embedded components but also the possibility to easily build CPS by using one, two or more boards, without changing programming model.

A Single Board Computer (SBC), named "AXIOM-Board", has been developed at the beginning of the 2017, (Figure 4.7), to build up a heterogeneous system, which could be able to combine ARM cores and enough programmable logic (FPGA) for significant acceleration, providing a platform that can be suitable for wide range of scenarios like Artificial Intelligence, Smart Home Living, Smart Video Surveillance, just to name a few.

In this paper, we will draw some conclusions of the main achievements of the AXIOM project, such as the programming model, based on a modified version of OmpSs [66] and the Data-Flow Threads (DF-Threads) execution model [106], [1], [23], [83], [52], [147], [50], [145], [179], [81], [180], [15]. We describe the fast link interconnect, named "AXIOM-Link", through which is possible to connect multiple boards using inexpensive cables, such as USB-C ones, while reaching up to 18-Gbps for each channel.

For the sake of completeness, we briefly recall here the main capabilities of our

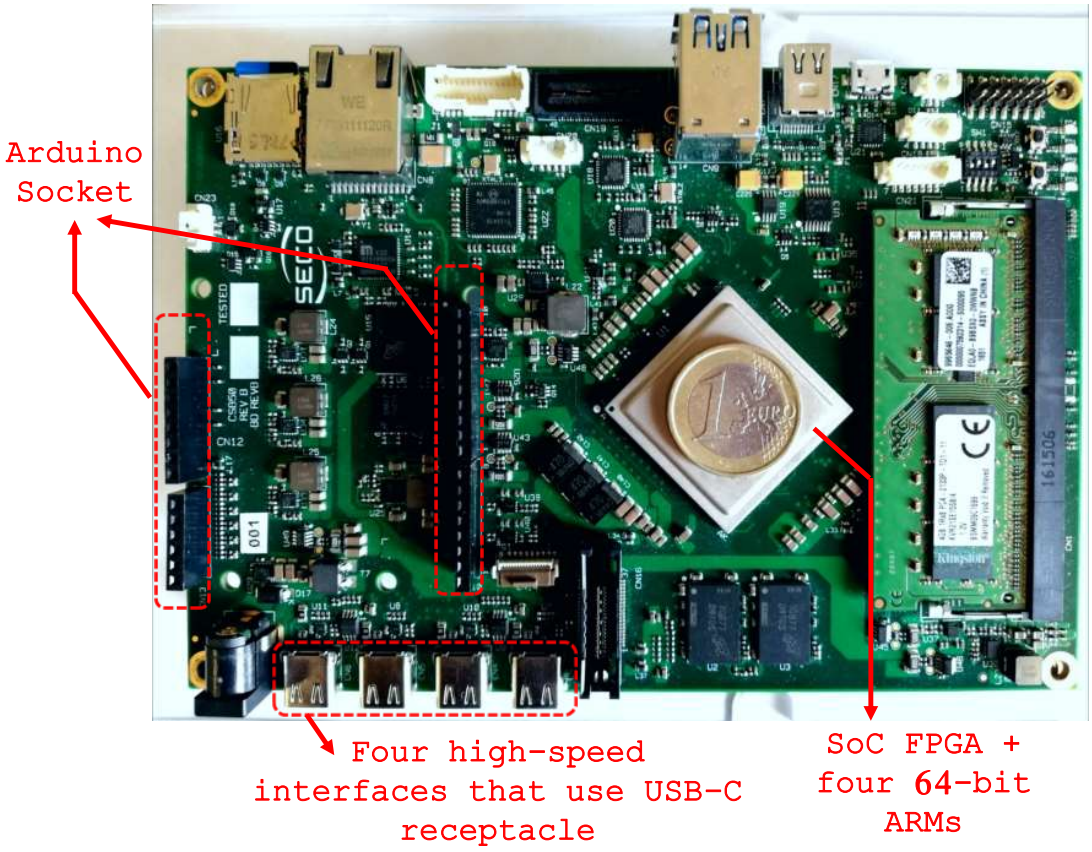

Figure 4.7: The AXIOM board based on an MPSoC Zynq Ultrascale+ (ZU9EG platform, expandable DDR4 memory up to 32 GiB, with four 2-lane 10Gbps gigabit transceivers that use USB-C cables, and an Arduino socket [39].

platform: the Xilinx ZU9EG platform [170] includes four 64-bit quad ARM Cortex-A53 General Purpose Processors (GPPs) working at a frequency of up to 1.5GHz; 32KB L1 Cache and 1MB L2 Cache. It can support several activities such as the OS (or system tasks) and whenever there is a sequential task that invokes Instruction Level Parallelism (ILP) rather than other forms of acceleration. Moreover, a Dual-Core ARM Cortex-R5 processing unit specialized for Real-Time tasks, working at a frequency of up to 600MHz; 32KB L1 Cache and 128KB of tightly coupled memory for each core.

The processors are encapsulated as part of the processing system (PS) as well as general-purpose interfaces such as two UART, two full-duplex SPI, and two full CAN 2.0B-compliant CAN, two USB, four 10/100/1000 tri-speed Ethernet, two I2C, ARM Mali-400 GPU, a Display Port, DDR4 Controller, 17-channel 10-bit ADC, and up to 128 GPIOs.

Furthermore, the PL covers up to approximately 300K LUTs, 32 Mb Block-RAMs, up to 2,520 DSP slices, and up to 24 bidirectional gigabit transceivers with a maximum

of 16.3 Gbps throughput. These transceivers are exposed to the physical world through the USB-C receptacle (using a custom AXIOM protocol, not the USB-C protocol), which is easier to be used due to its two-fold rotationally-symmetrical connector. Moreover, the board has a 250MHz trace port, which has also been used in other projects such as the H2020 HERCULES project [181].

One of the main goals of the AXIOM project was to design the hardware/software layers for multi-core, multi-board and heterogeneous system that has been envisioned by the project partners in order to fulfill the needs of future Cyber-Physical Systems (CPS). In this respect, the partners (BSC, EVIDENCE, FORTH, HERTA, SECO, University of Siena, VIMAR) identified use case scenarios, analyzed the AXIOM concept against possible exploitation paths, evaluated the AXIOM board to assess its capabilities. The inferred information has been translated into the definition of the AXIOM architecture (i.e., the need for an SoC with high-speed interconnects and FPGA), its external interface and its functional requirements.

For instance, modular scalability is enabled by a high throughput and low latency interconnect and the possibility to interface the applications directly to such high speed interconnect via reconfigurable hardware. Moreover, the Arduino UNO socket permits the use of a large set of tested so-called "shield" containing, e.g., relays, sensors and actuators. The Arduino software AVR-binary compatibility is ensured by a custom AVR soft-IP. This led to the choice of using the Xilinx MP-SoC Zynq Ultrascale+ (ZU9EG) platform [170]. Also, this choice opens the possibility to reach a design that is capable of being interfaced to the physical world and be used in smart applications where critical operation could be offloaded to the FPGA. The FPGA also provides a greater energy efficiency compared to executing the same function in software [182] and an appropriate substrate for the integration of our key features by providing customized and reconfigurable acceleration.

Therefore the design of the AXIOM hardware and software has been driven by the following pillars:

1. MP-SoC FPGA, i.e., The combination of large Programmable Logic (PL) with the ARM-based General Purpose Processors (GPPs), to support the Operating System (OS) and for running tasks that make little sense on the other accelerators,
2. Open-source software stack for a broader adoption,
3. Lower-Level Thread Scheduler for a higher predictability,
4. High-speed, inexpensive interconnects managed by an efficient Network Interface Card (NIC) [174],

5. Efficient interfaces for the Cyber-Physical world, such as Arduino [176] connectors (to be able to interface with sensors and actuators), USB, Ethernet.

4.6.2 Validating the AXIOM board against the COTSon simulator

Figure 4.8 shows our evaluation setup of two AXIOM boards interconnected via USB-C cables, without the need of an external switch. By using synergistically our framework and Vivado tool-chain, we synthesized DF-Thread execution model on Programmable Logic (PL).

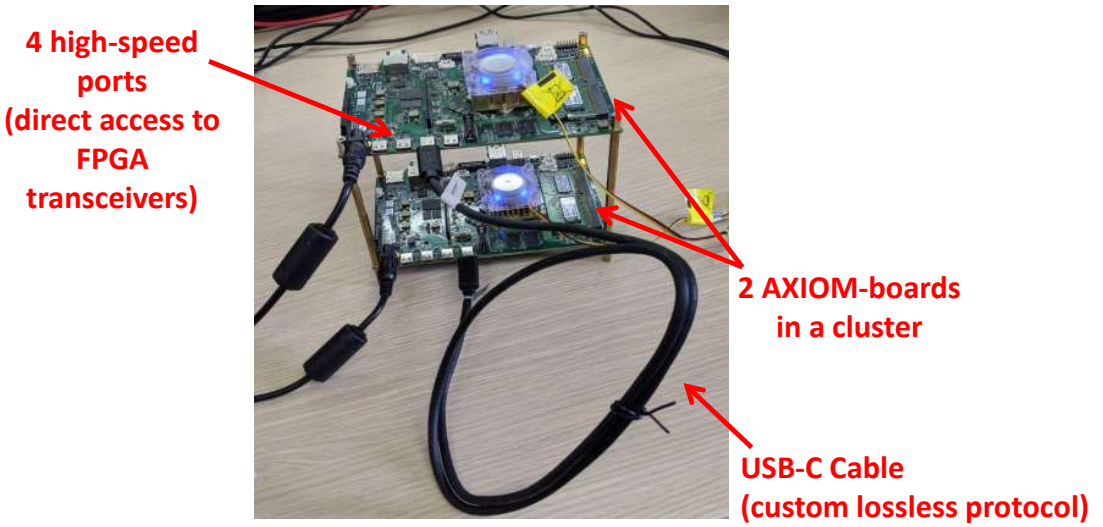

Figure 4.8: Two AXIOM boards interconnected up to 18-Gbps via inexpensive USB-C cables. The AXIOM board is based on a Xilinx Zynq Ultrascale + ZU9EG platform, four high-speed ports (up to 18-Gbps), an Arduino socket, and DDR4 extensible up to 32 GiB. As can be seen from the picture we do not need any external switch but just two simple USB-C cables to connect the two systems [57].

An important step in the design is to make sure that the design in the physical board is matching the system that was modeled in the COTSon simulator. As an example, we show in Figure 4.9 and 4.10 the execution time in the case of the BMM and RADIX-SORT benchmarks respectively, when running on the simulator and on the AXIOM board, while we vary the input data size. The timing are matching closely, thus confirming the validity of our approach. We scaled the inputs in such a way that the number of operations doubles from left to right (input size). In Figure 4.9), we have the BMM benchmark, where the input size represents the size of the square matrices, which are used in the multiplication. In Figure 4.10), we have the Radix-Sort benchmark, where the input size represents the size of the list to be sorted.

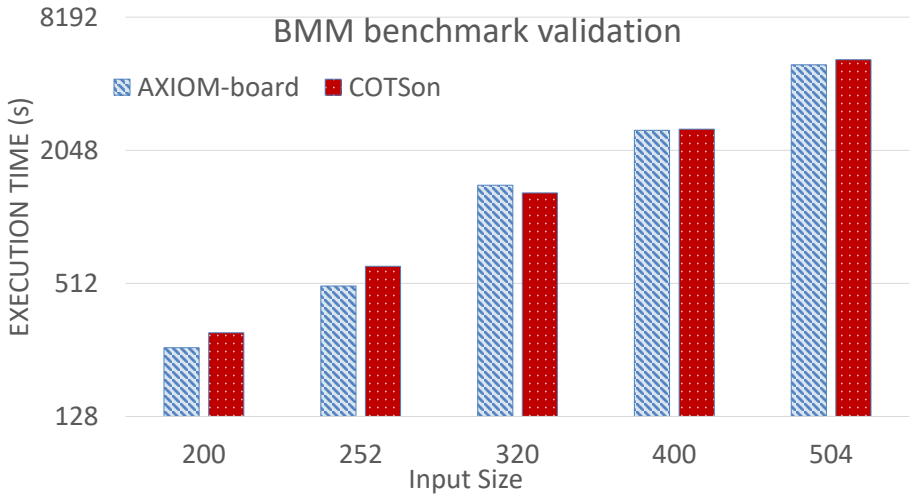

Figure 4.9: Validation of the execution time of the simulator against the AXIOM-Board. the Blocked Matrix Multiplication (BMM with different sizes (weak scaling). The results on the actual board match closely the simulations .

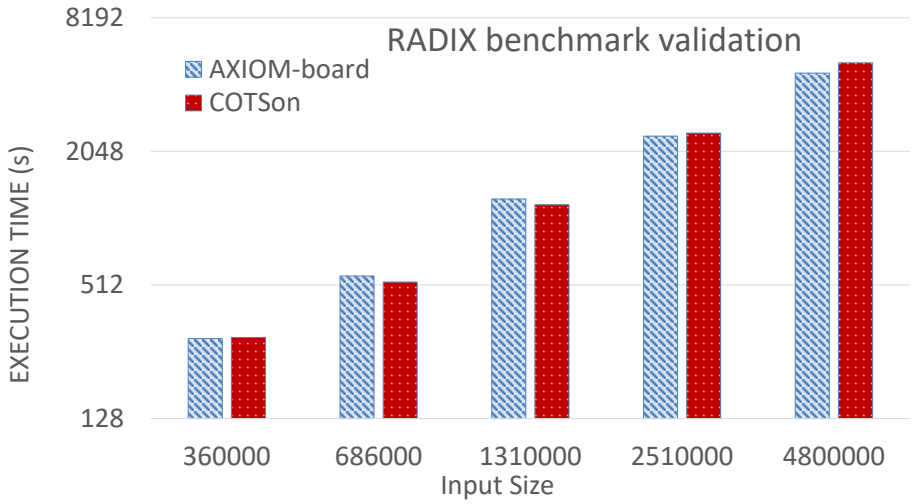

Figure 4.10: Validation of the execution time of the simulator against the AXIOM-Board. Radix-Sort benchmarks with different sizes (weak scaling). The results on the actual board match closely the simulations.

4.7 Final remarks

In this chapter, we presented our workflow in developing an architecture that could be controlled by the designer in order to match the desired key performance metrics. We found that it is very convenient to use synergistically the Xilinx HLS tools and the COTSon+MYDSE framework in order to select a candidate architecture instead of developing everything just with the HLS tools.

Thanks to the "functional-directed" approach of the COTSon simulator, we can de-

fine the architecture of any architectural components (i.e., a cache) for an early DSE and migrate to HLS only the selected architecture. Our DSE tool-set facilitates the modeling of architectural components in the earlier stages of the design. We have modified the classical HLS tool-flow, by inserting a modeling phase with an appropriate simulation framework, which can facilitate the architecture definition and reduce significantly the developing time.

We described the simple example of defining a two-way set associative cache through the timing model of COTSon. After, we illustrated the code migration from COTSon to Xilinx HLS tool, showing that the timing description made in the COTSon simulator is conveniently close to the final HLS description of our architecture. However, synthesizing of the HLS description of the cache design in Vivado HLS takes about four hours on a powerful workstation, while we were able to simulate it in COTSon in a few seconds.

By using the workflow presented in this paper, we were able to successfully prototype a preliminary design of our Dataflow programming model (called the DF-Threads) for a reconfigurable hardware platform leading to the AXIOM software/hardware platform, a real system that includes the AXIOM board and a full software stack of more than one million lines of code made available as open-source (https://git.axiom-project.eu/).

Chapter 5

FPGA Implementation of DF-Threads Co-processor

Programmable logic for the digital computing ecosystem was introduced back in the early 1960s to enhance the speed and power efficiency [183]. Thanks to the continual improvement in Very Large Scale Integration (VLSI) architecture design, FPGAs have undergone many million gate computing platforms, which offer reconfigurability and spatial programming logic ecosystem. Recently, this goes beyond and makes FPGA generations to be integrated with logic blocks, embedded high-throughput memory, fast routing switch bars, and more recently, microprocessors all on one single chip, so-called. System on a Chip (SoC) [170].

FPGAs are widely used in prototyping Embedded Computers. They more recently have become a significant component as the accelerators in the High-Performance Computing (HPC) and Cyber-Physical Systems (CPS) field, since they undertake tasks with higher reliability, reconfigurability, and energy efficiency [184]. Reconfigurable logic like FPGAs propose outstanding ways to boost specific functions but need enough tools in order to moderate the complicated programming [16], [185], [184].

Considering reconfigurable computing platform based on FPGAs, many works have offered solutions to address the issues of dynamic allocation of tasks for general-purpose-multi-core processors [180], or reconfigurable logic [186]. Nevertheless, these approaches have been effectively investigated only on single and multi-core superscalar architectures [187].

Modern FPGAs propose a dense area of logic building blocks, which lead to enhanced performance over power consumption that is also plausible with computational ASIC and the reconfigurability. FPGAs are suitable alternative for ASICs for reducing the time-to-market and those applications in which frequent update is necessitated.

Nowadays, FPGAs have established themselves in numerous applications, including aerospace and defense systems, Digital Signal Processing (DSP), Artificial Intelligence (AI), automotive, computer hardware emulation, and many others. In this section, we focus on literature addressing FPGAs in the context of high-performance computing.

In this perspective, to speed-up the parallel computing performance leveraging FPGAs, we need to consider two critical factors: available resources and scalability. The first increases each year with the addition of embedded high-speed memories (e.g., ultra RAMs), DSP blocks, and fast microprocessors. However, the demand for programmable resources is much higher as more complex systems are being deployed in high-performance computing. A well-known solution is to exploit parallelism in multiple FPGA platforms for a distributed system. In this ecosystem, logic reconfigurability, board-to-board communication, and design structuring become significantly sophisticated as the number of FPGA boards employed is increased. For instance, for CPU-FPGA heterogeneous systems, there can be many configurations, such as network topology, to construct an FPGA cluster. In such a system, sustained data transfer throughput between FPGA memory and CPU memory on a remote node is one of the important factors to decide a topology of the cluster. The study for clusters with a large number of FPGAs has not yet been fully investigated. Enhancements applied to systems with a small number of FPGAs are not plausible in systems with many devices. In general, factors like cost of system debugging, costs and challenges for data communication path, and concerning the clock distribution scheme, and reconfiguration among the boards are issues to be reviewed in such systems. Researchers yet to investigate the properties and applications for a cost-effective massive FPGA cluster framework for the given factors. Our research group to build a low-cost and easy to be interconnected (i.e., with USB-C cables) cluster of SoC FPGAs has proposed the Gluon board [188].

In the context of the AXIOM project [32], [178], [36], [37] it was realized that there is an extreme fragmentation of both devices and tools for embedded processing. Specifically, when more complicated functionalities are required, the entire system must be revised, and a new tool-chain must be adopted. Thus, our goal was to permit the programmers to deploy the device with a possible standard and open-source tool-chain based on a full Linux OS software distribution. However, scaling the performance of a computing system while retaining easy programmability is still on the headlines [189]. Additionally, tool-chains require to be integrated with suitable high-level synthesis tools in order to have a higher control of the programmable logic [190], [191].

Multi-processor system-on-chips (MPSoCs) are currently well-adopted, but the handling of many threads is still a source of many inefficiencies. Their management must

consider the order of execution, scheduling and the OS impact on threads executions (see Section 3.2). This aspect begins to be serious as the entire system grows in complexity, memory hierarchies, interconnects, and distributed resources. In this thesis, we propose to reduce such inefficiencies by using an efficient execution model named DF-Threads [1], [50], [106]. We explore the energy consumption of the AXIOM board for distributing DF-Threads across the AXIOM boards through a custom of board-to-board message types.

The work presented in this chapter has taken advantage of the recent growth in the number of parallel hardware components in recent MPSoC architectures (i.e., Zynq Ultrascale+). This model exploits all these resources efficiently compared to the conventional (i.e., von-Neumann) models and preserving the simplicity of the conventional programming models. Each block of threads in the proposed model comprises a set of sequential instructions that facilitate programmability while preserving the benefits of fine-grain parallelism coming from the Data-Flow distribution of blocks.

We choose the AXIOM board to build a cluster of MPSoC FPGAs as the hardware infrastructure to materialize the novel DF-Threads execution model. This chapter describes the proposed and implemented architecture in detail and evaluates its functionality and performance with the Recursive Fibonacci benchmark as a preliminary test for stressing the DF-Threads management.

5.1 Preliminary evaluation

In our research group, the proposed architectural support for DF-Threads execution has been verified and evaluated through the HP Labs COTSon Simulator [41], which allows us to find the best architecture setup of the heterogeneous multi-board platform (i.e., the AXIOM board). We used COTSon to define new hardware and software platforms easily and evaluate the scalability for heterogeneous multi-node architecture before spending time in real hardware implementation.

OpenMPI was selected for the programming model reference to provide a fair evaluation compared to our proposed programming and execution model (i.e., DF-Threads). OpenMPI is widely used in multi-node clusters to provide a parallel programming ecosystem. COTSon simulator can model the main components depicted in Figure 1.2. The DF-Threads Co-processor (DFC) is modeled into the COTSon simulator, and as a reference benchmark verifying the feasibility of supporting DF-Threads, a Blocked Matrix Multiplication (BMM) [192] is chosen, that is widely used as the kernel in recent smart applications such as machine learning.

As can be seen in Figure 5.1, the execution time improves, increasing the number of nodes/cores better than the OpenMPI. The normalized execution time gain is derived from the division of the execution time of the base-line, which is the OpenMPI, over the execution time of the DF-Threads.

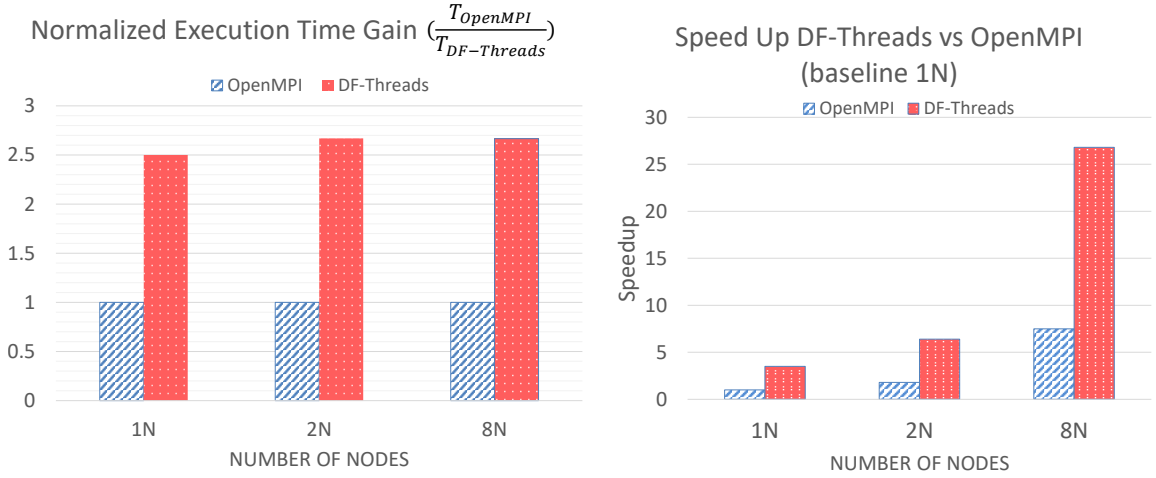

Figure 5.1: Comparing normalized execution time gains and speedup between the DF-Threads and the OpenMPI programming model running a Blocked Matrix Multiplication (BMM) with the block size of 8 and matrix size of 864 under the COTSon simulator [106]. The normalized execution time gain shows the execution time obtained from the OpenMPI as the base-line over the execution time of the DF-Threads.

The results presented in Figure 5.1 show that the DF-Threads execution model provides efficient scalability increasing cores/node counts. By adopting the DF-threads in the embedded domain (our target hardware platform), the system can be scaled-up by merely providing more computational nodes. These results motivated us to have a further implementation of simulated architecture on real-world hardware (the AXIOM board), through which we were able to prove the functionality and performance of the proposed DF-Threads Co-processor (DFC).

5.2 DF-Threads management

Recently, there has been an enormous exertion to move forward general programming models with thread management such as Cilk, OpenMPI. But in most of these models, synchronization and distribution of data between cores/nodes need to be managed manually by programmers and imposes an extra effort [171].

Instead, DF-Threads execution model proposes better scalability by re-managing the distribution of threads based upon the Data-Flow paradigm [1], [193]. It has capability of distributing the DF-Threads in a multi-node system. In order to have an efficient and scalable execution and movement of threads across the AXIOM board,

a low-level fine-grain data distribution technique based upon the DataFlow-Threads (DF-Threads) [1], [50], [106] modality has been adopted. DF-Threads make it possible to perform a more predictable real-time execution since a thread's input data is made available before the thread execution, so the time to execute a thread can be estimated very precisely.

In particular, the DF-Threads execution model relies on multi-threading, on which dynamic Data-Flow principles are applied to shape a Data-Flow graph (DFG) among threads and to exploit Control-Flow execution inside a thread efficiently. Moreover, fetching the instructions ready to execute is deterministic (when all inputs of the thread are available), which makes it near to optimal since the DF-Threads management schedules the life-time of each thread and knows which thread will be executed at any time. Starting from the proposed DF-Threads [1], in this study, we design DF-Threads Co-processor (DFC) providing the scheduling and management of the threads for heterogenous multi-node systems. DFC relies on the Load Balancing Unit (LBU) to manage the DF-Threads distribution either locally or remotely. The LBU distributes the thread to other boards when local resources finish, by dispatching appropriate packet descriptors to the Network Interface Controller (NIC) [174] to bypass the software stack.

Figure 5.2 shows the block designs exploited to materialize and map the DFC on the PL part of the Zynq U+ platforms as an individual soft IP. DF-Threads Co-processor (DFC) is tightly coupled (i.e., based on the AXI Stream protocol and proper buffering) to the NIC [174] module to be able to transceive appropriate messages in order to distribute the workloads among the network. Since the DFC is offloaded on the PL, all overheads regarding the thread management are reduced. In order to avoid the overheads and costs of proprietary existing communication protocols a NIC [174], which was implemented during the AXIOM project, has been adopted. It permits to achieve an efficient and scalable program execution and a seamless interconnection of systems spanning multiple boards [178]. Such connectivity permits users to expand and scale-up their system by easily interconnecting more boards flexibly and low-costly, without the need for particular expensive cables, connectors, or external switches.

AXIOM board has four bi-directional links providing different network topologies such as ring, torus and 2D-mesh, etc. The AXIOM routing algorithm is based on the store-and-forward packet transmission with Virtual Circuits (VCs). A discovery process is initiated at power-up by the master node to fill up the routing tables by dedicated node IDs. As such, all the packets will be transceived through the physical links based on corresponding information stored in the routing table.

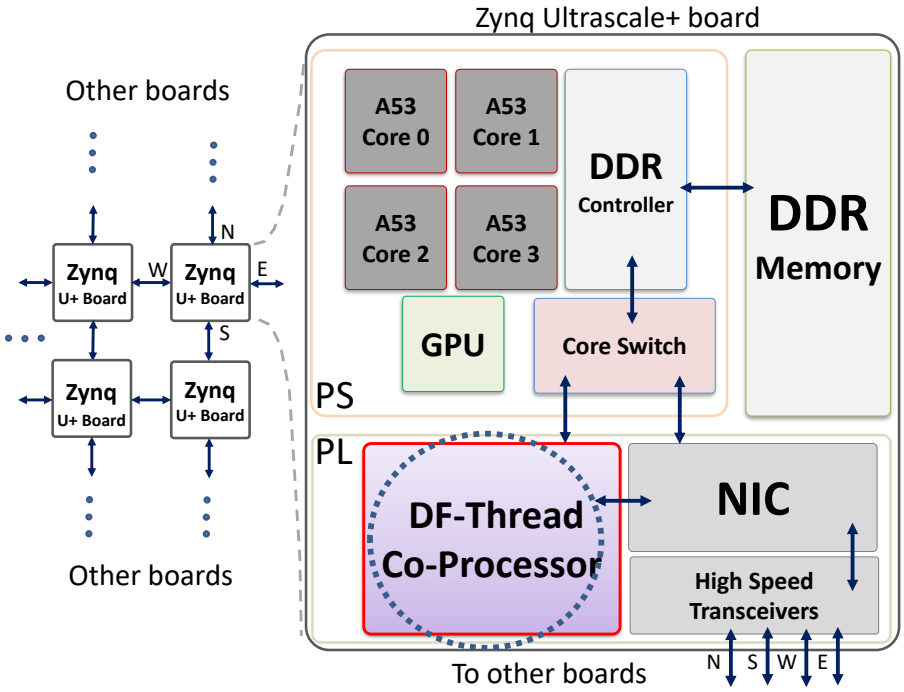

Figure 5.2: Proposed scalable DF-Threads architecture mapped on the Zynq Ultrascale+ based board like AXIOM board (left side). The detailed bock designs for each node (board) are depicted at the right side. The proposed DF-Threads scheduler is completely designed on PL (dotted circle). U+: Ultrascale+, PS: Processing System, PL: Programmable Logic, NIC: Network Interface Card [174].

Some functions cannot be performed in software as they have an execution overhead, which is too large. The DF-Threads Scheduler serves to offload the management of the thread descriptors either locally or across the boards. thread descriptors are meta-data that are specific for each DF-Threads, such as Frame Pointers (FPs), Intstruction Pointers (IPs), and Synchronizaion Counters (SCs), which specify that when a thread can be executed. The DF-Threads that become ready to execute are stored in a queue so-called DF-Threads Ready Queue (DFRQ) (See Section 5.5). The central component of the DFC, DF-Threads Scheduler (DFS), coordinates the asynchronous execution of threads under the Data-Flow Graph (DFG) of producer and consumer threads. The DFC is responsible for managing the threads descriptors, from when a DF-Threads is born, executed, and finally destructed. DFS knows when a DF-Thread is ready to be executed by tracking the SC values when a new input is available. When all the inputs become available, the DF-Thread is stored into the DFRQ and is ready to be executed. During the life of a DF-Thread, only those that are ready to execute are allowed to be distributed; otherwise, they stored locally.

5.3 Introduction to API

The DF-Threads execution is based on the producer-consumer semantic, through which a DF-Thread (consumer) is allowed to execute only when all its inputs are produced by other DF-Thread(s) (producers). A combination of hardware/software is deployed hierarchically to manage the DFC entirely implemented on the PL. At the top level (API level, not the programmer), a set of instructions are used to decide the life-time of each DF-Thread, and how the outputs know the address of the other threads inputs. Classical Data-Flow architectures were proved to be inefficient in terms of supporting data structures and procedural calls [194]. Our DF-Threads is proved to provide more efficient support for a full range of language features (e.g., arrays, data structures) [147], [82], [179]. The life-time of a DF-Threads is determined through four API calls [147], which are reported in Table 5.1.

Table 5.1: DF-Thread APU in a C like syntax used in software stack of the proposed MPSoC FPGA cluster. Uint64_t and uint8_t are fixed width types as defined here, for exemplification here., but Can be overloaded with any base type of a, e.g., 64-bit machine (The table partially is adopted from the [1].

API Instruction	***Description***
void* df-schedule(void* ip, uint64_t sz, uint8_t sc);	A data frame with the size of "sz", and "sc" number of inputs will be allocated, and the start address of it will be returned as "Frame Pointer (FP)" . "ip" is the Instruciton Pointer to the pointer of where the function instruction is located. The DF-Threads will not be executed until its "sc" becomes "0" (all its inputs are available).
void* df-load();	Loads the data from the (itself) input frame
void* df-write(void* fp, uint64_t val);	fp (frame pointer) points to where the data "val" is stored. It is assumed that writes are snooped by the architecture (in particular by the DF-Threads co-processor - DFC), so that, for every input that is written, the "sc" of the DF-Threads to which the "fp" points is decremented.
void* df-clear(void* fp);	Frees memory pointer to by "fp", and clear the Data Frame associated to the DF-Thread. The DFC tracks the deallocated memory.
Void* df-decrease(void* tfp, uint8_t sc_n)	For the frame pointed by "tfp" , which is a pointer to the target frame, its stored "sc" field into the metadata part of the frame will be decreased by the value of "sc_n".

This API is the interface between the co-processor located on the PL and the compiler and run-time system building run-time libraries. This API provides more flexibility to be adopted for our MPSoC FPGA cluster. Programmers can access memory re-

gions with specific allocating semantics after scheduling a DF-Threads via ***df-schedule*** function. Allocation of all required frames is done before running the application, and the DFC creates a list of FPs (Frame Pointers). For each DF-Thread a portion of memory is associated so-called ***frame-data***. Each ***frame-data*** consists of two sub-regions named ***meta-data-region*** and ***data-region***. Calling ***df-schedule***, dequeues an available FP from the allocated list, and performs filling the meta-data of the frame. The DFC handles either locally and remotely (in a distributed manner), the meta-data (also called continuations: like ***ip***, ***sc***, ***sz***), and the information about the core or node on where the thread is born, or running. ***ip*** is the instruction pointer of associated operation to the frame, ***sc*** is the synchronization counter which counts the number of available inputs by the producer, and ***sz*** is the size of the frame in bytes.

As described in Table 5.1, a DF-Thread lifetime is determined via a light-weight API. For the simplicity of the easier mapping of a generic program code, DF-Threads are considered to be simple C functions without the need to use the entire stack for passing parameters (token like or continuations), additionally to Data-Flow paradigm [195]. Listing 5.1 illustrates the creation (or scheduling) of a DF-Thread with explicit management of the input frame (where input data is stored). Starting from this API, an opcode is assigned to each of the DFC instructions, which are passed to the DFC through the registers resided between the PS and the PL (See Figure 5.4). The API communicates the DFC through a well-optimized and specialized linux device driver, which maps the DFC device tree including registers into the kernel-space.

```
1  /* This code is a C like code that examplificates
2     how a DF-Thread life-time is created.*/
3  void df-thread (uint64_t *fp){
4    df-load();
5    //<statement_1>;
6    //<statement_2>;
7    //.
8    //.
9    //<statement_n>;
10   df-clear(fp);
11 }
```

Listing 5.1: Lifetime for a DF-Thread in C

memory model Two listing codes are provided to present how a user code can be transformed into the code using DF-Threads API. Listing 5.2 shows simplified Recursive Fibonacci in C code, and Listing 5.3 presents the DF-Threads API used to manage the life-time of the DF-Threads. In Listing 5.2, there are two main operations; ***fibo*** and ***add***, two of which are translated into two separate functions deploying DF-Threads API. In this case, the ***df-schedule*** determines how many inputs the next instances will receive.

The ***df-write*** writes the frames inputs of the next instances. Once all inputs of the target DF-Thread (written into its associated target ***frame-data***) have been produced and written, the ***sc*** of target frame becomes zero, which means that target DF-Thread is ready to fire (executable). Finally, the current DF-Thread (running) is done, and its occupied ***meta-data*** into the shared memory should be cleared, which is performed through the ***df-clear(fp)***. Thanks to the proposed API, the use of a standard compiler (e.g., GCC) is possible for producing the binary for the target platform (e.g., x86, and AArch64).

```
/* This code is a C like code of
a Recursive Fibonacci*/
int fibo (int n) {
if (n <= 1) return n;
return fibo(n-1)+fibo(n-2);
}
```

Listing 5.2: "C" like code for a Recursive Fibonacci example

```
/* How a DF-Thread life-time is created
   for a Recursive Fibonacci function.*/
void fibo(void) { // DF-Thread Fibonacci
  uint64_t* myfp =(uint64_t*) df-load();
  int n = myfp[1];
  if (n <= 1) {
    df-write(myfp[0],n);
  }else {
    uint64_t* tfib1 = df-schedule(&fibo,2);
    uint64_t* tfib2 = df-schedule(&fibo,2);
    df-write(tfib1[1], n-1);
    df-write(tfib2[1], n-2);
    uint64_t* tadd = df-schedule(&adder,3);
    df-write(tadd[0], &myfp[0]);
    df-write(tfib1[0], tadd+1));
    df-write(tfib2[0], tadd+2);
  }
  df_clear();
}
void adder(void) {
  uint64_t* myfp =(uint64_t*)df-load();
  uint64_t f1 = myfp[1];
  uint64_t f2 = myfp[2];
  df-write(myfp[0],f1+f2);
  df-clear();
}
```

Listing 5.3: Transformed code of Recursive Fibonacci C code into the DF-Threads API code

5.4 Memory model

A Distributed Shared Memory (DSM) is deployed to provide remote memory accesses needed by the producer and consumer threads executing on different nodes (boards). A Global Address Space (GAS) [51] is defined for all nodes, for each of which has assigned a fixed portion of the address space.

The need for traditional data coherency is not requested thanks to the single-assignment rule inherited to Data-Flow nature for producer-consumer manner. For the DF-Threads execution, various memory types are articulated in shared and non-shared portions to store the programs' data and meta-data. This model enables the implementation of four types of a distributed shared-memory system (even in non-coherent cases) in the context of producer-consumer as briefly described in the following [1].

- ***N-to-1***: In this model, there are N DF-Threads producing N outputs that will be later consumed by only one DF-Thread. To implement this pattern, a portion of the distributed shared memory is allocated so-called a ***Frame Memory (FM)***.
- ***1-to-1***: This pattern is used for "self commutation," which means that the DF-thread is using a large portion of dynamically allocated private memory. It is called ***Private Memory (PM)***.
- ***N-to-N***: There are N producing output DF-Threads, that N other DF-Threads will consume their result. The atomic transactions between a distributed shared memory cells are required. The mechanism is handled through a region in memory, so-called ***Transactional Memory (TM)*** [79], [84], [196]–[198].
- ***1-to-N***: It happens when a DF-Thread produce a result for multiple locations of the memory possibly consumed by multiple DF-Threads. The communication is managed through the FM, as a typical case, when several consumers consume an array. This case is managed by distributing pointers for the element located in another shared portion, so-called Owner Writable Memory (OWM).

In this Chapter, we rely on the ***N-to-1*** memory for the current implementation of the DFC, as depicted in Figure 5.3. The other memory types have not been yet exploited on our proposed distributed hardware architecture based on the Zynq Ultrascale+ and

are considered to be used for this study's future work.

The Distributed Shared Memory (DSM) presented in Figure 5.3 allows us to access any producer-consumer DF-Threads across the network. An allocator assigns a fixed portion of the shared memory as ***Frame Memory (FM)*** to each node, which can access to ***FM*** of each other remotely. Each ***FM*** consists of two regions: ***Meta-data Region***, where the thread descriptors are stored, and ***Data Region***, where the associated inputs/outputs frame data are resided.

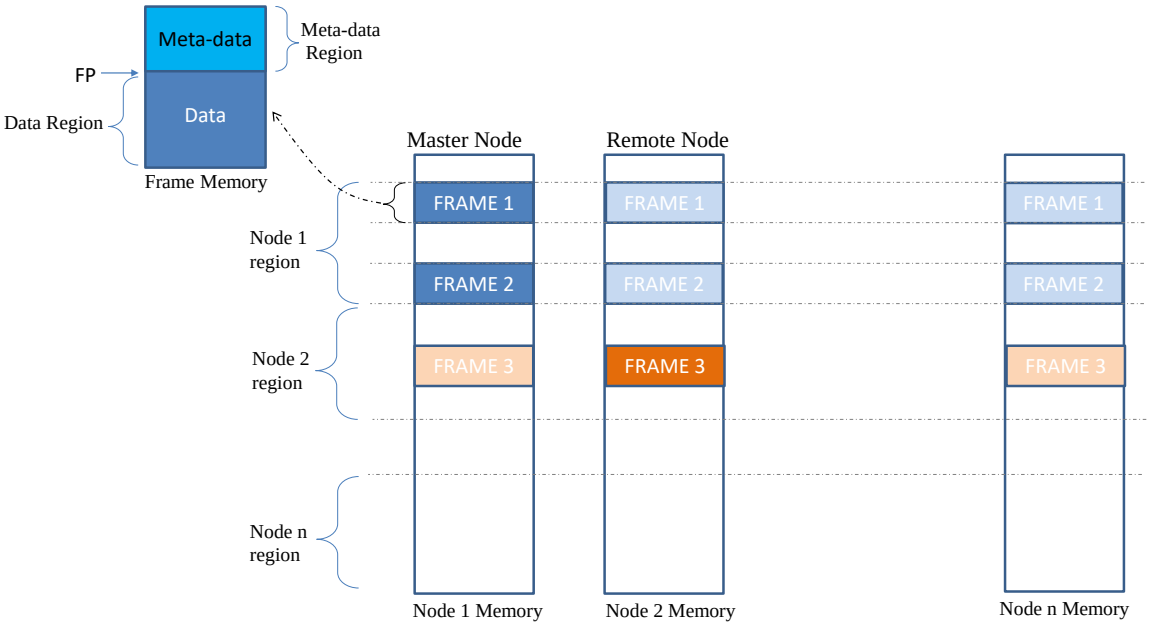

Figure 5.3: The deployed Distributed Shared Memory (DSM) region, showing the associated ***Frame Memory (FM)*** allocated for each DF-Threads. The allocator for each node in the network dedicates a separate portion of the memory, which is accessible by other remote nodes. Each FM comprise to regions: ***Meta-data*** and ***Data***, to where is pointed by the ***Frame Pointer (FP)***. ***Meta-data*** is a tiny portion storing the DF-Threads continuation meta-data.

5.5 Architecture block diagram

In this section, we briefly explain the design of the DF-Threads Co-processor (DFC) fully implemented on the Programmable Logic (PL) of an MPSoC FPGA (e.g., The AXIOM board) (See Figure 5.4). The detailed explanation of each block IPs is currently under the publication. The design is connected to the Processing System (PS) through the AXI HPM (High-Performance Master) port with the 128-bit data width. This interface is based on the AMBA AXI4 protocol and is made suitable for high-performance communications. It provides separate address channels supporting burst transactions

and issuing multiple outstanding addresses with out of order responses.

An AXI Interconnect IP (Courtesy of Xilinx IP libraries) splits the 128-bit data width into the 64-bit data width between two modules: the DFC and the customized System Timer, which is used to measure the performance key metrics of the system. The direct communication between the DFC and the cores resided in PS is made through a set of AXI-lite memory-mapped registers implemented on the PL.

The API of the DF-Threads execution model passes its arguments (i.e., opcode, argument1, and argument2) through a set of ***Registers*** resided on the PL. Most of the important configuration of the co-processor is parameterized. ***Registers*** is intended to be used to initialize and set reconfigurable parameters of the essential constituent sub-modules. For example, the size of the frame into the memory that is associated with each DF-Threads can be configured through the ***Registers***, or the masks, pending and clear registers of the IRQ Handler module (Interrupt Handler). The IRQ handler takes care of the exceptions and interrupts coming from the local FIFOs, and sub-modules, which finally performs a logical "AND" on all its inputs producing a one-bit interrupt signal connected to the PS. The received instructions from the PS is decoded through the ***Decoder***, which is also a splitter of streams of instructions into their corresponding Finite State Machines (FSMs). Notably, all PL sub-modules interfaces are based on the AXI4-Stream protocol with 64-bit data width, which provides proper high-performance communication. AXI4-Stream is a point-to-point efficient interface for high-speed interconnections without the use of AXI addresses. Each AXI4-Stream performs in a single unidirectional data channel (Data-Flow path), including handshake side-channels.

For the current implementation of the memory model, the address space of the off-chip DDR memory of the AXIOM board is divided into two parts: Global Memory (GM) region, where the distributed shared ***Frame Memory (FM)*** resided, and the DF-Threads Ready Queue (DFRQ) region, where the DF-Threads that are ready to be executed are located. A ***Memory Management Unit*** is specialized to handle the data exchange between the PL sub-modules and the PS DDR. This unit provides multiple access (WR/RD) between the PL sub-modules and the PS-DDR through the slave High-Performance Coherent (HPC) port, which is based on the AXI4 interface.

A round-robin arbitration policy for Memory Management Unit is selected to arbitrate among multiple input requests from different processes inside the PL. The Load Balancing Unit (LBU) performs the distribution of DF-Threads among the remote nodes through the NIC TX (RX) Controllers, which compose (interpret) the sent (received) descriptors to (from) the NIC module [174]. A unique ID so-called "Node ID" (NID) is

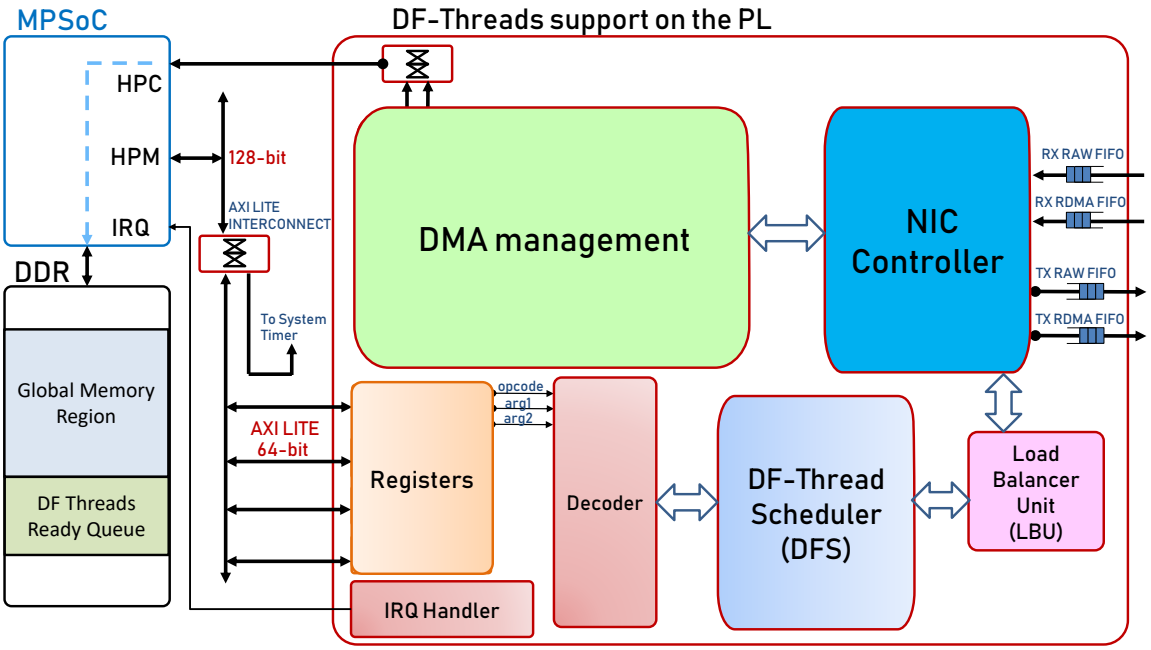

Figure 5.4: Simplified block diagram of the architectural support for the DF-Threads Co-processor (DFC) fully implemented on the PL of an MPSoC FPGA platform (e.g., the AX-IOM board). NIC: Network Interface Controller, DMA: Direct Memory Access, HPC: High-Performance Coherent, HPM: High-Performance Master.

assigned to each node of the network that is obtained from the NIC module. The NID aims to identify the source and destination (producer and consumer) of the traversing DF-Threads.

The tool that is used to develop these modules is Xilinx High-Level Synthesis (HLS) tool, Vivado Design Integrator tool. The Xilinx Petalinux 2016.3 is deployed to generate the Board Support Packages, including device tree, Linux kernel, and First Stage Boot Loader (FSBL). As described in Figure 4.1, the XGENIMAGE tool, which is a wrapper of the Bootstrap tool is used to generate the bootable image dumped into a micro SD card.

The Xilinx Vivado HLS tool allows us to use a higher level of abstraction written in C/C++. Therefore, we decided to use C++ to benefit from some useful C++ libraries (like data streaming support). Thanks to the HLS directives/pragma option, we were able to optimize the designed IPs in terms of the clock cycle latency and the resource utilization for our target platform (i.e., Zynq Ultrascale+). Moreover, We used Xilinx System Development Kit (SDK) to test the functionality of the entire. PL design as a co-processor for PS before running the design under the Petalinux. It allows us to eliminate the time consuming work-flow of BSP and OS image generation while yet work with the designed PL modules from the bare-metal stand-alone software libraries provided by the Xilinx SDK.

We adopted the following methodology for each HLS IP modules separately. During the development of the HLS IPs, we first implemented the preliminary draft of the module in C++ and then compiled it to generate the output RTL. We analyzed the clock cycle latency and intervals of the generated RTL by applying timing constraints to the design (clock period equal to 6.5 ns). With a further investigation on some modules, we noticed that the best directive that can be deployed notably is PIPELINE, enhancing both the clock cycle latency and initiation interval. We wrote a basic program running under the SDK environment to test and debug during the PL design development. Furthermore, three Xilinx ILA (Integrated Logic Analyzer) IPs are attached to the DFC on the PL to probe the crucial signals of the design involved in the performance key metric definitions.

The AXI4-Stream interface is the main data stream interface that is used for the high-speed data paths. In each HLS module, the desired interfaces have appeared as AXI4-Stream ports by defining specific directives through the specialized pragmas. The definition of input in HLS based on an AXI4-Stream interface is shown in Listing 5.4). Moreover, the interface of all the used FIFOs is based on the AXI4-Stream with 64-bit data width. The utilized HLS IPs specifically use data streaming libraries (***hls_stream.h***) provided by HLS to facilitate the interfacing based on the AXI4-Stream. For instance, reading from an "AXI4-Stream FIFO" is shown in Listing 5.5).

```
#include <ap_int.h>
#include <hls_stream.h>
typedef ap_uint<64> uint64;
struct axi4_stream64_t{
  uint64 data;
  uint1 last;
};
void hls_module (stream<axi4_stream64_t> &input) {
#pragma HLS INTERFACE axis off port = input
}
```

Listing 5.4: The definition of an AXI4-Stream interface in HLS

```
struct axi4_stream64_t{
  uint64 data;
  uint1 last;
};
axi4_stream64_t stream_buffer;
void hls_module (stream<axi4_stream64_t> &input) {

  //<statement_0>
  //<statement_1>
  //.
  if(!input.empty()){
```

```
12      stream_buffer = input.read();
13    }
14    //.
15    //<statement_n>
16  }
```

Listing 5.5: Reading from an AXI4-Stream FIFO

5.5.1 The Decoder

We select one component (The "Decoder" module) of the DFC as an example, and describe it in this sub-section to show that how we have implemented the IP modules. This block is a Multiplexer, controlled by the "op_code" (it is defined as an input) specifying a particular operation of the DFC. The "Decoder" splits out the received "arg1" and "arg2", which are the arguments of DFC instructions, receiving from the "Instructions Registers." The algorithm flowchart presenting the operations carried out by the "Decoder" is drawn in Figure 5.6. As can be seen in Figure 5.4, the "Decoder" module decodes the received "op_code" and provides the arguments for the operations performed by the corresponding Finite State Machines (FSMs), including schedule, decrease, fetch, and clear.

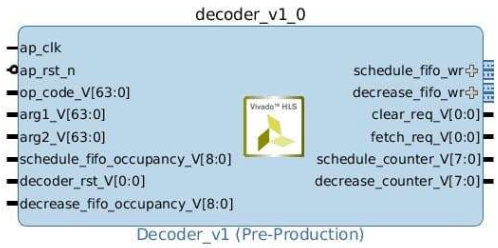

Figure 5.5: Top view of the ***Decoder*** module designed in HLS.

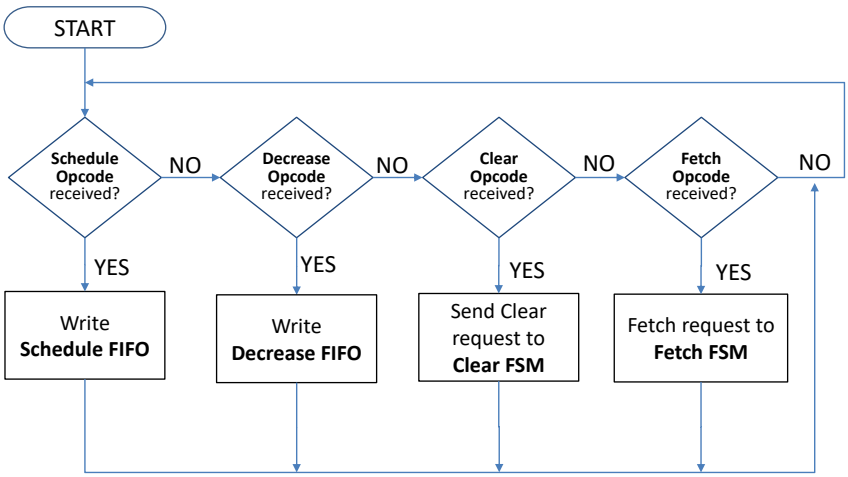

Figure 5.6: Flowchart showing the algorithm of ***Decoder*** module written in HLS.

Figure 5.5 shows the top view of the "Decoder" module implemented in HLS. The description for essential inputs and outputs are provided in Table 5.2. Moreover, the timing diagram of the input and output signals of "Decoder" module is derived by the ILA (Integrated Logic Analyzer) and is depicted in Figure 5.7, which shows a latency of three clock cycles. The plot is prepared through the WAVEDROM tool [199]. The

design is optimized in such a way that it only consumes two PL clock cycles to generate the outputs.

Table 5.2: Input and output signals of the ***Decoder*** module briefly described.

IO name	***Type***	***Description***
op_code_V[63:0]	IN	The received unique "opcode" dedicated for each instruction (schedule, decrease, fetch, clear) from the "Instruction Registers".
arg1_V[63:0]	IN	The first argument received from the "Instruction Registers". The composed fields of this 64-bit input varies in different instructions. For instance, in "df-schedule" API, the first argument is an "Instruction Pointer (IP)" .
arg2_V[63:0]	IN	The second argument of the desired instruction received from the "Instruction Registers"
decoder_rst_V[0:0]	IN	The active high signal to reset the internal FSM of the module including all variables.
schedule_fifo_occupancy_V[8:0]	IN	The number of bytes occupied by the "schedule FIFO" . "Decoder" module uses this data to stop writing to the "schedule FIFO", when reaches to a certain level of occupied bytes.
decrease_fifp_occupancy_V[8:0]	IN	The number of bytes occupied by the "decrease FIFO" . "Decoder" module uses this data to stop writing to the "decrease FIFO", when reaches to a certain level of occupied bytes.
schedule_fifo_wr[63:0]	OUT	The AXI4-Stream channel to the "schedule FIFO"
decrease_fifo_wr[63:0]	OUT	The AXI4-Stream channel to the "decrease FIFO"
clear_req[0:0]	OUT	The clear request to the "clear FSM"
fetch_req[0:0]	OUT	The fetch request to the "fetch FSM"
schedule_counter_V[7:0]	OUT	For measurement usage: it counts each "df-schedule" instruction.
decrease_counter_V[7:0]	OUT	For measurement usage: it counts each "df-decrease" instruction.

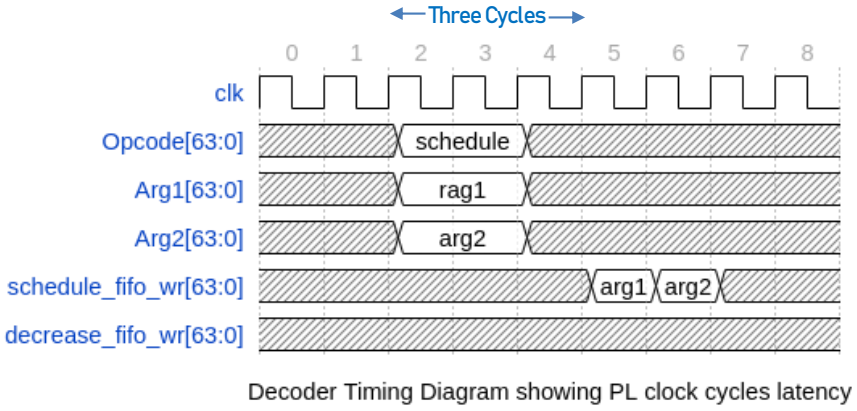

Figure 5.7: The timing diagram of the input and output signals of the ***Decoder*** module.

5.5.2 The Load Balancing Unit (LBU)

The efficient distribution and scheduling of tasks in a distributed system is yet a compromising effort and open research topic [200]. A well-known and practical method of this kind of parallel computation is "work-stealing" [201], in which processors require stealing computational tasks from other processors. In the full software version of such schedulers, if the problem's granularity is not big enough, the overhead arising from the distributing task, and the communication is comparable and reduces the efficiency of the processing power [202]. In this section, we present a dynamic Load Balancing Unit (LBU) fully implemented on the Programmable Logic (PL) of a Xilinx Zynq Ultrascale+ board (AXIOM board)[39], [58]. In contrary to work-stealing policy, the LBU performs dynamic load balancing based on a semi-work-stealing method, where a light handshake mechanism is established in a producer-consumer manner to distribute the DF-Threads across the nodes.

In this implementation, the distribution of the DF-Threads is dicoupled from the execution of DF-Threads by offloading the communication patterns and load balancing, which eases the processor's burden. Each DF-Threads executes in sequential on the cores located into PS. Certain instructions to be executed are associated with the memory regions so-called ***Data Frame***, which their executions initiate once all of its inputs are available. The inputs can be provided by other DF-Threads executing locally or remotely. Instruction pointers of those frames that are ready to be executed (FPs) are stored in a circular buffer called Dataflow-Threads Ready Queue (DFRQ).

The LBU is responsible for distributing the Frame Pointers (FPs) associated with the ready DF-Threads to available remote nodes. The load balancing algorithm differs from the pure work-stealing, the stealer node, that asks for the DF-Threads will get the available threads only when the overloaded node permits the threads migration. Those nodes in which their DFRQ is "Almost Empty" send an "LB-QUERY" message to their neighbors. This message will be propagated through the cluster until one node has a DFRQ with the "Almost Full" flag enabled. This node sends an ACK message to the node requesting for threads, and then the requester node will accept the available DF-Threads by an ACCEPT message (See Figure 5.8). Finally, the node which provides DF-Threads will perform an RDMA (Remote Direct Memory Access) [178] to migrate a certain number of FPs to the requester node. The number of migrated FPs, and DFRQ thresholds (Almost Empty, Almost Full) are set through the "Registers" dedicated to this module.

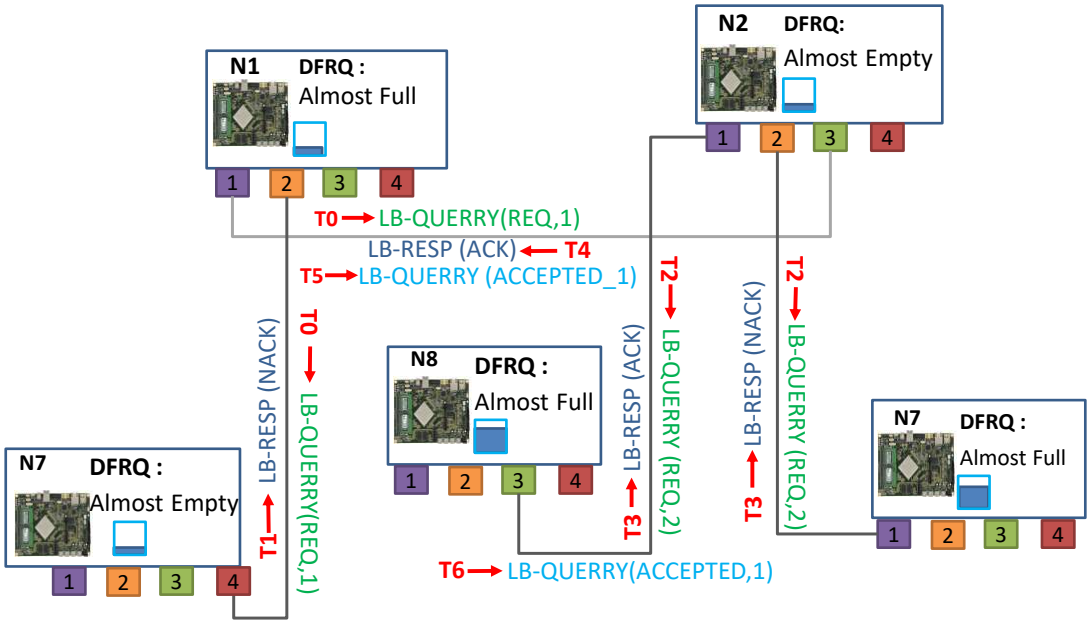

Figure 5.8: A simple example to show the load balancing query message passing among the AXIOM boards. DFRQ: DF-Threads Ready Queue, LB-ACK: Load Balancing Acknowledgement, LB-NACK: Load Balancing Negative Acknowledgement.

5.6 Experimental results

5.6.1 Recursive Fibonacci benchmark

We use a Recursive-Fibonacci benchmark on the two nodes (the AXIOM boards) as the initial exploration and to stress the capability of managing many threads of our proposed implemented DFC. The main function of the benchmark is FIB(n,th), in which "n" is the size of Fibonacci and "th" is a threshold stopping the generation of the parallel recursive calls. Different input size with a constant threshold has been evaluated. The Load Balancing Unit (LBU) parameter has been set appropriately for each of the input sets, following the total required DF-Threads in respect to the input size.

The presented result in Figure 5.9 proves the implemented DFC proper functionality and a reasonable degree of scalability. The result of the proposed DFC on two-node is convincingly aligned with the results obtained in the COTSon simulator, confirming that the LBU properly distributes a set of many DF-Threads across the multi-board MPSoC FPGAs. As shown in Figure 5.9, an increase in the input size increases the number of generated DF-Threads, and consequently, the speedup increases due to the more efficient exploitation of parallelism. In this experiment, the threshold of Fibonacci is kept constant while the input size is being increased for creating different size of parallelism providing a fair evaluation in terms of speedup. Moreover, the threshold and the max-

imum number of the migrated DF-Threads by the LBU is kept constant to adapt with the simulator experiments, and observe its functionality in respect with different input size. The experiment shows that the LBU is performed more efficiently for a larger input size. It is worthwhile to mention that it the first time that the DF-Threads execution model is implemented on a real multi-board hardware platform, and the results are valid compared with the results of the COTSon simulator.

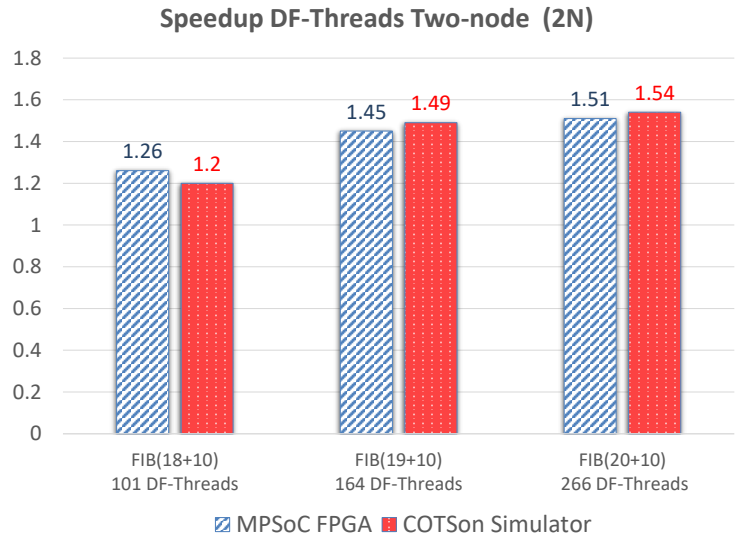

Figure 5.9: Comparing speedup of the Recursive Fibonacci benchmark with different input sizes (e.g., FIB(20+10), 20 is the input size and 10 is the threshold (i.e., where the generation of the parallel recursive calls will be stopped) running under the DF-Threads co-processor on two nodes (real implementation against COTSon simulator). The number of DF-Threads that are generated are shown below each FIB examples.

5.6.2 Resource utilization

The resource utilization of the implemented DFC is derived from the Vivado 2016.3 after the placement and routing phase (it is reported in Table 5.3). It should be noted that the utilization for Block RAM tiles is mainly due to the intermediate buffers used internally between the various modules (e.g., between the Memory Management Unit and the PL sub-modules). We decided to optimize the implemented HLS IPs in such a way to keep the size of utilized BRAM tiles as low as possible. Almost no BRAM tiles have been utilised for the DFC assembled sub-modules excluding the LBU, where two table (one for the TX path, and the other for the RX path) are used for storing the status of the LB-QUERRY. As can be seen in Table 5.3, the total utilization of the DFC is nearly 11%, which makes it a flexible IP module to be easily ported in smaller FPGA platforms in terms of the size of the CLBs.

Table 5.3: Resource utilization of the proposed DF-Threads co-processor fully implemented on the PL (the AXIOM board). CLB: Configurable Logic Blocks. CLB: Configurable Logic Block, LUT: Look Up Table.

PL Module	*CLB LUTs*	*CLB Registers*	*CLB*	*LUT as Logic*	*LUT as Memory*	*LUT as Flip Flop pairs*	*Block RAM Tile*
Registers	1399	1805	419	1399	0	548	0
Decoder	231	211	83	231	0	45	0
DF-Threads Scheduler	899	1790	347	899	0	346	0
DMA Management	1752	2120	423	1654	98	784	0
NIC Controller	1108	1458	276	1108	0	620	0
Load Balancing Unit	1484	1957	356	1484	0	393	2
Total	30985	43094	7164	28724	2261	14171	64.5
Available	274080	548160	34260	274080	144000	274080	912
Utilization (%)	11.3	7.86	20.91	10.48	1.57	5.17	7.072

5.6.3 Power consumption analysis

A reliable and precise method to measure and monitor the power consumption of the system is necessary in order to enable optimization towards the energy efficiency. Additionally, the ability to estimate power consumption in a design is mandatory for efficient part selection and system reliability. Referring to the AXIOM board, there are specifically dedicated eight INA219 power monitor integrated circuits to monitor the crucial power rails of the board, and are reported by Table 5.4. These INA219 ICs communicate with the FPGA through an I2C bus connected to the PS, and more detailed information on the INA219 can be found in [203].

In order to extract power values for the crucial rails of the board, we performed the experiments while DFS issues the RAW and RDMA messages of 1000M length in 10 cycles. The duration of the test was 240s with 200ms sampling time. Essentially, the total power consumption of the board remained between 1W and 1.6W (sum of the seven crucial power rails). Figure 5.10 and 5.11 illustrates the maximum power variations for the crucial voltage rails during RAW and RDMA transactions respectively.

As can be seen from Figure 5.10 and 5.11, the MGTAVTT voltage rail has the highest power consumption since the gigabit transceivers' termination circuits with 1.2V supply voltage sink larger amount of current. The average power consumption in client mode has 10.15% larger value in comparison with server mode due to an extra processing effort to re-compose the acknowledge message and send it back to the server. Moreover, since in our DFS implementation we did not utilize any access to the PL DDR (we access to the PS DDR), the average power consumption for the 1V2_DDR_PL voltage rail remained below 5.5mW.

Table 5.4: AXIOM board's power supply rail adopting dedicated power monitors.

Power supply rail	*Nominal Voltage [V]*	*Description*
VCC_INTFP	0.85	PS full-power domain supply voltage
VCCINT	0.85	PS internal power supply
INTFP_DDR	0.85	PS DDR controller and PHY supply voltage
1V2_DDR_PS	1.2	PS DDR supply
1.2V_DDR_PL	1.2	PL DDR supply
MGTAVCC	0.9	Analog supply voltage for GTH transceiver
MGTAVTT	1.2	Analog supply voltage for GTH transceiver termination circuits

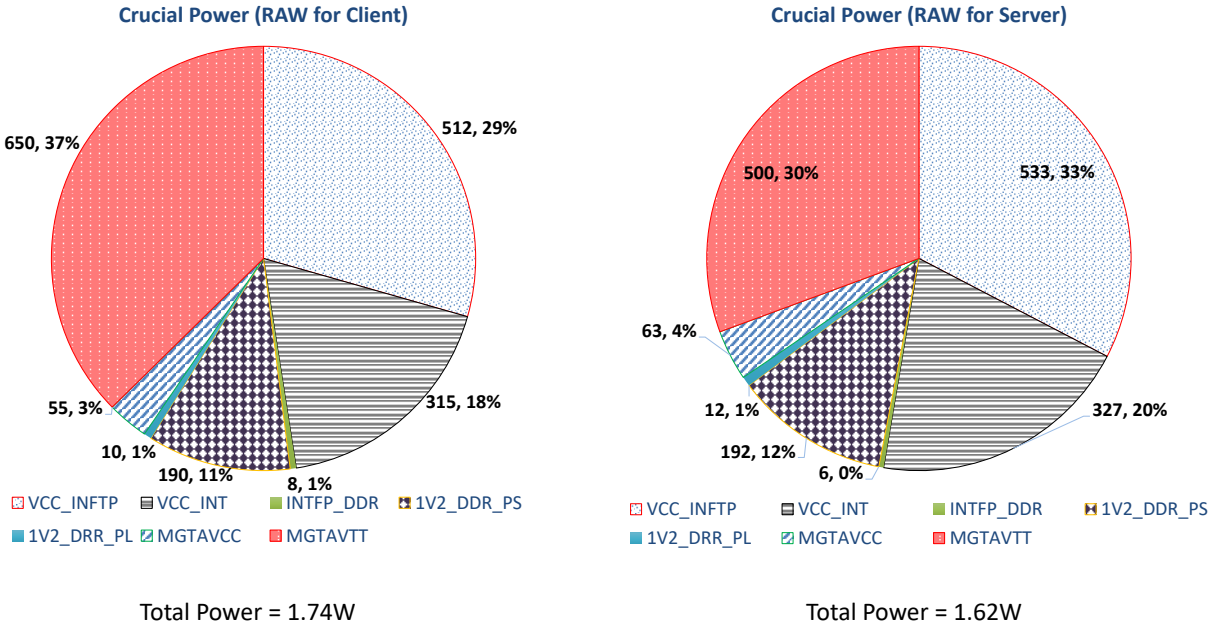

Figure 5.10: Power consumption budget for the Client board (left) and the Server board (right) when NIC RAW messages are issuing.

Finally, comparing power consumption between RAW and RDMA message, the RDMA message type consumes in average 9.7% less than the RAW message types. This arises from the extra dedicated logics to deal with data of RAW messages while for RDMA messages, the data are efficiently moved to the PS DDR by using the Xilinx Data Mover soft IP.

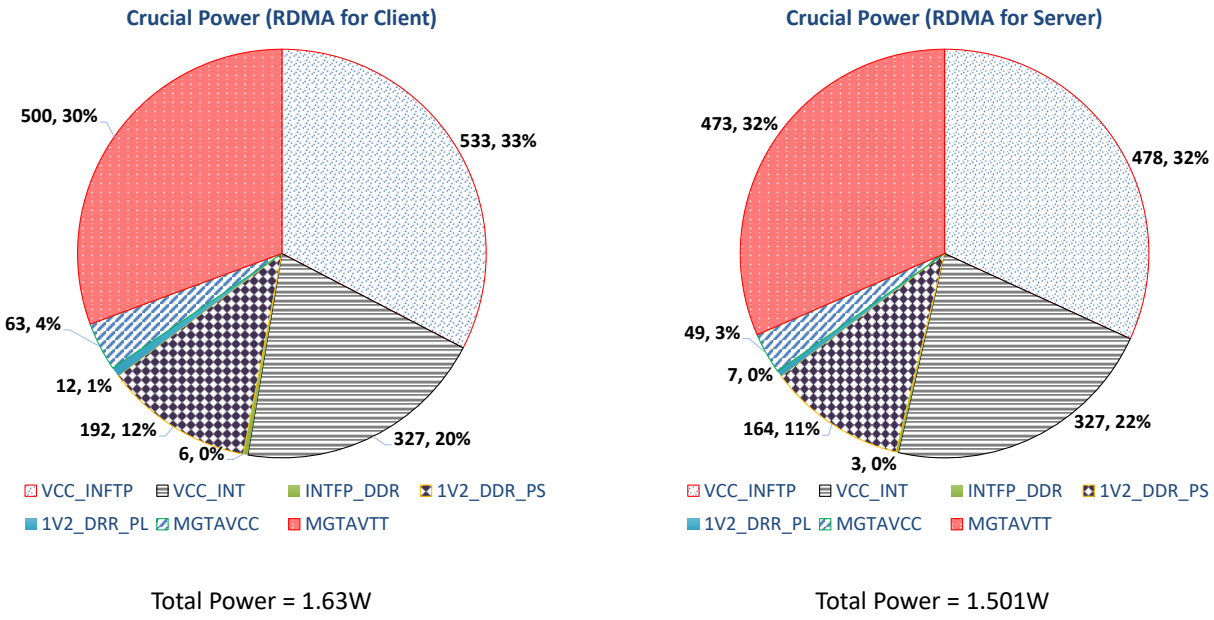

Figure 5.11: Power consumption budget for the Client board (left) and the Server board (right) when NIC RDMA messages are issuing.

5.7 Final remarks

This chapter presented a scalable implementation of the DF-Threads execution model deployed on a heterogeneous distributed system composed of the AXIOM boards. We proposed a DF-Threads co-processor (DFC), which permits efficient scalability across the boards by using a load balancing protocol. Finally, we analyzed some power measurements in the context of the AXIOM project as well.

We briefly described the DFC hardware implementation on an MPSoC FPGA (the AXIOM board). Starting from a light-weight programming model handling DF-Threads life-time previously proposed by our research group, we were able to build the micro-architecture of the DFC on the FPGA instance. A Recursive Fibonacci benchmark has been chosen as a preliminary test to validate the implemented DFC proper functionality compared with the COTSon simulator, and also for stressing the thread management of the DFC.

The implemented design is in an early stage and it can be optimized to reach a better performance. For instance, we noticed that there is a latency between the PL sub-modules and the "DFRQ Region" into the PS DDR. As future work, to cope with this issue, we need to deploy hybrid regions between the PS DDR and the PL BRAMS for storing the Frame Pointer (FP) of those DF-Threads that are ready to execute. As such, the memory accesses will significantly decrease due to the available local data for the PL DF-Thread Scheduler and Load-Balancing unit boosting the overall performance of

the DFC. In order to be able to optimize the efficiency of the board, we explored the energy consumption of the critical voltage rails while transceiving is performed across the nodes by the DF-Threads DFC.

Chapter 6

Conclusions and future works

This study aims at exploring an alternative solution for exploiting the scalability and parallelization of heterogeneous multi-node architectures, where efficient exploitation of exposed parallelism is yet a challenging effort [204]. Several causes are affecting an efficient execution, such as synchronization waits and memory latencies. The Data-Flow threads (DF-Threads) execution model offers exploited parallelism of the available on-chip resources by scheduling many parallel fine-grain threads and distributing them across processing elements of a multi-node system [50], [1], [106].

6.1 Summary

The work, which is presented in this thesis is divided into three sections. First, (in Chapter 3) we present our Design Space Exploration (DSE) tool-sets [40], [146]. The DSE is one of the principal phases of the design development before digging into the hardware prototyping of our novel DF-Threads execution model. This tool-set was also used during the TERAFLUX project [51] to design and evaluate complex architectures for High-Performance Computing (HPC) with 1000 general-purpose cores. Thanks to these tools, architectures can be analyzed in different aspects, including using various types of programming models (e.g., CILK++, OpenMPI), Operating System impact, or execution models (i.e., Data-Flow and von-Neumann models).

The DF-Threads execution model is adopted targeting a distributed heterogeneous multi-node architecture offered by the AXIOM board. The AXIOM board is a single-board computer (SBC) based on a Zynq Ultrascale+ FPGA, and it is capable of being interconnected with high-speed (up to 18 Gbps) inexpensive cables (i.e., the USB3 cables). A preliminary design of the DF-Threads execution model, which was previously modeled through the DSE tool-sets, was validated on the AXIOM board.

As a further implementation of the proposed DF-Threads, a methodology has been

deployed to translate the already verified model through the DSE tool-sets and COTSon simulator to the AXIOM board (in Chapter 4). The COTSon timing model is gradually migrated into the High-Level Synthesis (HLS) tool. The performance of the migrated design is validated and aligned with the outcome of the COTSon simulator. Thanks to this validation, the COTSon simulator allows us to rapidly explore the design space before mapping it on real-world hardware. A further study is carried out to explore the operating system's impact on the performance, thanks to the COTSon simulator, which offers a full system simulator. It has been shown that the percentage of the kernel overheads can reach up to 60% of the total execution time, comparing four different Linux Distributions (See Section 3.2).

In Chapter 5, the memory model deployed for the cluster of SoC FPGA board (AXIOM board) is described. The model is based on the distributed shared memory (DSM) model, in which a specific portion of the shared memory has been allocated to each node. The proposed building blocks of the co-processor, implemented on the Programmable Logic (PL) of the SoC FPGA, is presented. One of the critical building blocks of this architecture is the Load Balancing Unit (LBU) module, through which the DF-Threads are distributed across the nodes of the cluster. The load balancer is designed specifically for Data-Flow-Threads (DF-Threads) and can support multi-node computing architectures. The LBU is highly parallelized with separate queues and FSMs dedicated for each TX and RX path to the custom network interface leading to the efficient management of DF-Threads.

We relied on the Recursive Fibonacci benchmark to study the performance of the implemented DFC module in managing and distributing the DF-Threads. The results show the proposed DF-Threads functionality and its capacity to distribute and manage many fine-grain threads across a two-node setup of the AXIOM board cluster. For the first time in this study, we present the feasibility of the DF-Threads execution model on the MPSoC FPGA platforms connected through the high-speed network interface cards. As shown in Section 5.6.1, the achieved results are reasonably aligned with the results obtained from the COTSon simulator. Finally, the power consumption of the essential voltage rails of the board was explored. This measurement is carried out when the LBU asks RAW and Remote Direct Memory Access (RDMA) messages to the Network Interface Card (NIC) [174] module, which is also utilized on the Programmable Logic (PL).

6.2 Future Work

After a further analysis made during the evaluation of the implemented DF-Threads co-processor, we noticed that there are some rooms for further design improvements. Figure 5.9 shows relatively good scalability of the model and the proper functionality. However, we need to evaluate it with more cores (up to four are available on a Zynq Ultrascale+ FPGA) and more real nodes (e.g., four, eight boards) interconnected.

The number of benchmarks used to evaluate the design is limited in this study due to the limitation in implemented memory model, which is only the N-to-1 model (See Section 5.4). There is a need to expand the memory model's implementation to other types, such as 1-to-N relying on a memory semantic so-called Own Writable Memory (OWM), allowing us to have other realistic benchmarks as the Matrix Multiplication.

Furthermore, software load balancers will provide enough performance as long as the number of threads is big enough compared to the load balancing overhead. To mitigate this overhead, delegating load balancing to an accelerator will improve the performance of such architectures. To draw a comparison and measure the overhead of the software based load balancing unit, a LBU will be emulated into the simulator.

Finally, we realized that there are a plenty of rooms for optimizing the PL sub-modules implementation, such as a hybrid DFRQ region between the PL and the PS DDR, or cache injection technique for passing the ready DF-Threads into the cache from the PL through the Accelerator Coherent Port (ACP) port of the PS, instead of passing through the PS-PL High-Performance Master interfaces (HPM). Moreover, we desire to employ an open-source RISCV processor soft IP replaced as the PS system to eliminate the critical path and introduced delay between the PS and the PL.

Appendix A

Publications

This study appeared in several conference papers, workshops and journals. They are listed in the following subsections.

Journal papers

1. R. Giorgi, **F. Khalili**, M. Procaccini, "Translating Timing into an Architecture: The Synergy of COTSon and HLS (Domain Expertise—Designing a Computer Architecture via HLS)", *International Journal of Reconfigurable Computing, Hindawi*, 2019. **High level synthesis (HSL) designs and algorithms, IPs integrations, and FPGA implementations.**:

Peer reviewed conference papers

1. R. Giorgi, M. Procaccini, **F. Khalili**, "AXIOM: A scalable, efficient and reconfigurable embedded platform", *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages:480–485, 2019. **FPGA implementation of the DF-Threads execution engine**.

2. R. Giorgi, M. Procaccini, **F. Khalili**, "A design space exploration tool set for future 1k-core high-performance computers", *Proceedings of the Rapid Simulation and Performance Evaluation: Methods and Tools*, pages:1–6, 2019. **Validation outcomes of the simulator against the equivalent FPGA-based design**.

3. R. Giorgi, M. Procaccini, **F. Khalili**, "Analyzing the impact of operating system activity of different linux distributions in a distributed environment", *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages:422–429, 2019. **Simulator discussions and feature comparison**.

4. R. Giorgi, M. Procaccini, **F. Khalili**, "Energy efficiency exploration on the zynq ultrascale+", *2018 30th International Conference on Microelectronics (ICM)*, pages:48–

54, 2018. **Demonstration of possible DF-Thread architecture on a cluster of Zynq Ultrascale+, power consumption measurements**.

Workshops and Summer schools

1. A. Sahebi, **F. Khalili**, G. Mariotti, M. Procaccini, and R. Giorgi, "Gluon-B: a Modular Board for FPGA Clusters", *5th Italian Workshop on Embedded Systems- IWES*, presentation, 2021. **Programmable Logic**.

2. **F. Khalili**, R. Giorgi, "A Dynamic Load Balancer for a Cluster of FPGA SoCs", *HiPEAC Summer School - ACACES*, pages:23–26, 2020. **Proposing, design, and implementation**.

3. **F. Khalili**, and R. Giorgi, "A DF-Threads Scheduler Co-Processor for a Cluster of FPGA SoCs", *HiPEAC*, Student Poster Session, 2019. **design and implementation of the load balancing unit**.

4. **F. Khalili**, R. Giorgi, "A Soft-IP for Performance Measuring of the Zynq Ultrascale+ CPU/FPGA interface", *HiPEAC Summer School - ACACES*, pages:5–8, 2019. **Proposing, design, and implementation**.

5. **F. Khalili**, M. Procaccini, R. Giorgi, "Reconfigurable logic interface architecture for cpu-fpga accelerators", *HiPEAC Summer School - ACACES*, pages:13–16, 2018. **Proposing, design and implementation**.

6. M. Procaccini, **F. Khalili**, R. Giorgi, "An FPGA-Based Scalable Hardware Scheduler For Data-Flow Models", *HiPEAC Summer School - ACACES*, pages:12–15, 2018. **FPGA Prototyping**.

7. M. Procaccini, **F. Khalili**, R. Giorgi, "An FPGA-based Scalable Hardware Scheduler for Data-Flow Models", *3rd Italian Workshop on Embedded Systems- IWES*, presentation, 2018. **Systematic design**.

Appendix B

Notation and Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
AXIOM	Agile eXtensible Input Output Module
BMM	Matrix Multiplication
CPS	Cyber-Physical System
CPU	Central Processing Unit
DDDP	Distributed Data-Driven Processor
DDM1	Data Driven Machine1
DDR	Double Data Rate
DFC	DF-Threads Co-processor
DFG	Data-Flow Graph
DFL	Data flow Language
DFRQ	DF-Threads Ready Queue
DF-Threads	Data-Flow Threads
DLP	Data Level Parallelism
DMA	Direct Memory Access
DSE	Design Space Exploration
DSM	Distributed Shared Memory
DSP	Digital Signal Processing
DTS	Distributed Thread Scheduler
ETS	Explicit Token Store
FF	Flip-flop
FIFO	First-In, First-Out
FP	Frame Pointer
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GM	Global Memory
GPP	General Purpose Processor
GPU	Graphic Processing Unit
HDFM	Hughes Data Flow Multiprocessor
HDL	Hardware Description Language
HLS	High Level Synthesis
HPC	High Performance Computing
HPCP	High-Performance Coherent Port
HPM	High-Performance Master
ID	Irvine Dataflow langugage
ILA	Integrated Logic Analyzer
ILP	Instruction Level Parallelism
IP	Instruction Pointer
IPs	Intellectual Properties
LAU	Language Assignation Unique
LBU	Load Balancer Unit
LUT	Look Up Table
MDFA	McGill Dataflow Architecture
MPI	Message Passing Interface
MPSoC	Multi-Processor System on Chip
NIC	Network Interface Card
NOC	Network on Chip
OS	Operating System
OWM	Owner Writable Memory
PCI	Peripheral Component Interconnect
PE	Processing Element
PGAS	Partitioned Global Address Space

PL Programmable Logic
PS Processing System
PU Processor Unit
RAM Random Access Memory
RDMA Remote Direct Memory Access
RFIB Recursive Fibonacci
RTL Register Transfer Level
SBC Single Board Computer
SBDT Software Briged Data Transfer
SHL Smart Home Living
SISAL Streams and Iterations in a Single Assignment
SVS Smart Video Surveillance
TCAM Ternary Content-Addressable Memories
TDDA Tagged-Token Dataflow Architecture
TDDP TI Distributed Data Processor
TDFL Textual Data-Flow Language
TLP Thread Level Parallelism
VAL Value-oriented Algorithmic Language
VC Virtual Circuits
VLSI Very Large Scale Integration

Appendix C

Vivado Design Blocks

The building blocks of the architectural support for the DF-Threads execution model, including the DFC, NIC, and the PS is shown in Figure C.1 implemented on the AXIOM board's MPSoC FPGA (Zynq Ultrascale+). The sub-modules assembling the DFC is depicted in Figure C.2.

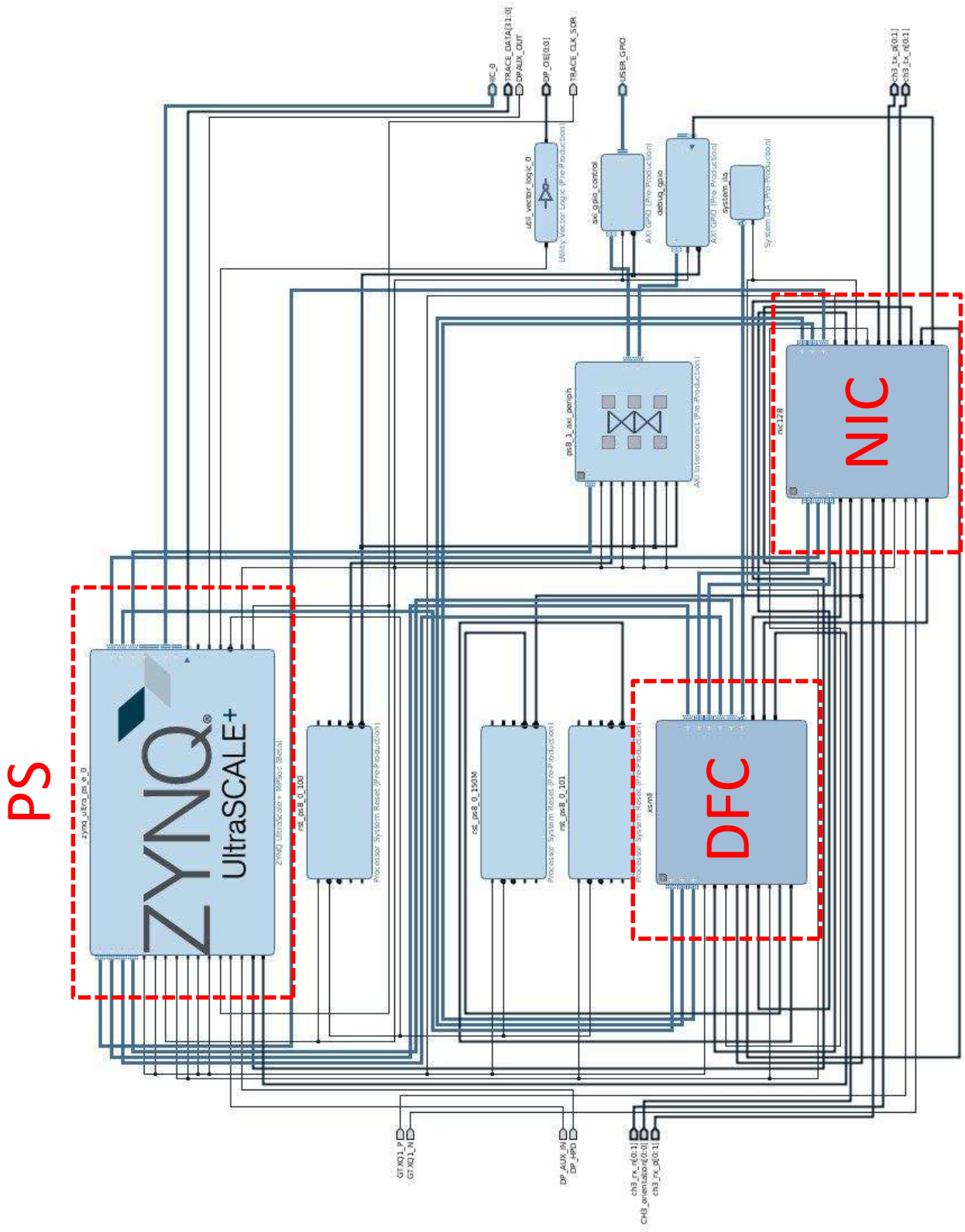

Figure C.1: The main building blocks of the MPSoC FPGA design on the Zynq Ultrascale+ supporting the DF-Threads execution model: DF-Threads Co-Processor (DFC), Processing System (PS), ad Network Interface Card (NIC) [174].

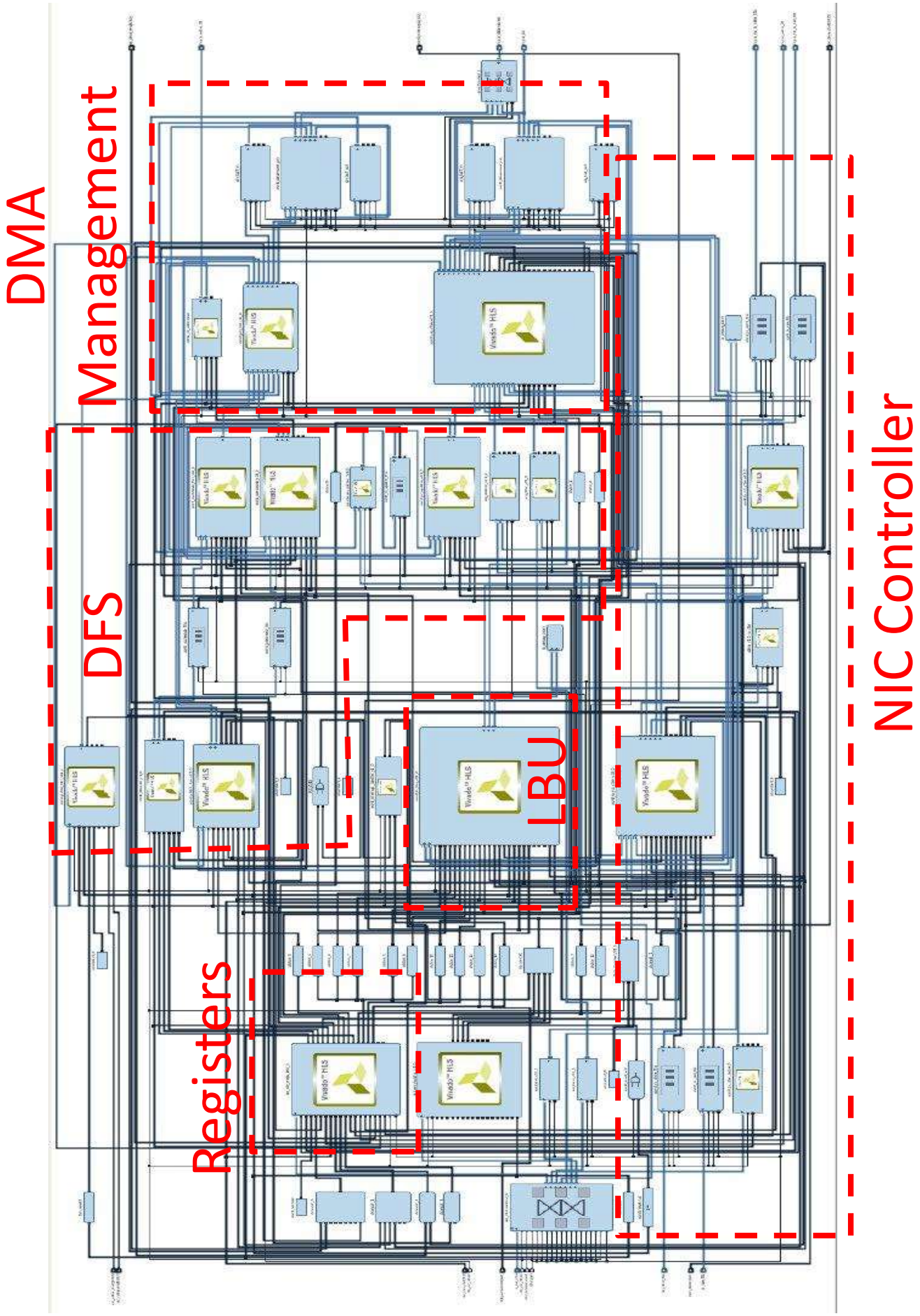

Figure C.2: DF-Threads Co-Processor (DFC) fully implemented on the PL. DFS: DF-Threads Scheduler, LBU: Load Balancing Unit.

Bibliography

[1] R. Giorgi and P. Faraboschi. An Introduction to DF-Threads and their Execution Model. In *Proceedings of IEEE International Symposium on Computer Architecture and High Performance Computing Workshop*, pages 60–65, Paris, France, 2014.

[2] R. Giorgi. AXIOM: A 64-bit reconfigurable hardware/software platform for scalable embedded computing. In *Proceedings of IEEE Mediterranean Conference on Embedded Computing (MECO)*, pages 113–116, Bar, Montenegro, 2017.

[3] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[4] Hanan Shukur, Subhi Zeebaree, Rizgar Zebari, Omar Ahmed, Lailan Haji, and Dildar Abdulqader. Cache coherence protocols in distributed systems. *Journal of Applied Science and Technology Trends*, 1(3):92–97, 2020.

[5] Julie Dumas, Eric Guthmuller, César Fuguet Tortolero, and Frédéric Pétrot. Trace-driven exploration of sharing set management strategies for cache coherence in manycores. In *2017 15th IEEE International New Circuits and Systems Conference (NEWCAS)*, pages 77–80. IEEE, 2017.

[6] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[7] ARB OpenMP. Openmp application program interface version 4.0, 2013.

[8] Charles E Leiserson. The cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[9] Samuel H Fuller and Lynette I Millett. *The Future of Computing Performance: Game Over or Next Level?* National Academy Press, 2011.

[10] Wesley M Johnston, JR Paul Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM computing surveys (CSUR)*, 36(1):1–34, 2004.

[11] 48 years of microprocessor trend data. `https://github.com/karlrupp/microprocessor-trend-data`. Accessed: 2020-12-20.

[12] J. Dennis and David Misunas. A preliminary architecture for a basic data flow processor. In *ISCA*, 1974.

[13] Jack B Dennis. Data flow supercomputers. *IEEE computer*, 13(11):48–56, 1980.

[14] Krishna M. Kavi, Bill P. Buckles, and U. Narayan Bhat. A formal definition of data flow graph models. *IEEE Transaction on Computers*, pages 940–948, 1986.

[15] A. Mondelli, N. Ho, A. Scionti, M. Solinas, A. Portero, and R. Giorgi. Dataflow Support in x86-64 Multicore Architectures through Small Hardware Extensions. In *Proceedings of IEEE Euromicro Conference on Digital System Design (DSD)*, pages 526–529, Madeira, Portugal, 2015.

[16] Walid A. Najjar, Edward A. Lee, and Guang R. Gao. Advances in the dataflow computational model. *Parallel Comput.*, 25(13-14):1907–1929, December 1999.

[17] Leandro AJ Marzulo, Tiago AO Alves, Felipe MG França, and Vítor Santos Costa. Couillard: Parallel programming via coarse-grained Data-Flow compilation. *Parallel Computing*, 40(10):661–680, 2014.

[18] Cor Meenderinck and Ben Juurlink. Nexus: Hardware support for task-based programming. In *Proceedings of IEEE Euromicro Conference on Digital System Design (DSD)*, pages 442–445, Oulu, Finland, 2011.

[19] Jeronimo Castrillon, Rainer Leupers, and Gerd Ascheid. Maps: Mapping concurrent dataflow applications to heterogeneous mpsocs. *IEEE Transactions on Industrial Informatics*, 9(1):527–545, 2013.

[20] Richard Townsend, Martha A Kim, and Stephen A Edwards. From functional programs to pipelined dataflow circuits. In *Proceedings of the ACM International Conference on Compiler Construction*, pages 76–86, Austin TX, USA, 2017.

[21] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: a functional data-parallel IR for high-performance GPU code generation. In *Proceedings of IEEE International Symposium on Code Generation and Optimization (CGO)*, pages 74–85, Austin TX, USA, 2017.

[22] Sebastian Ertel, Justus Adam, and Jeronimo Castrillon. Supporting Fine-grained Dataflow Parallelism in Big Data Systems. In *Proceedings of the ACM International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 41–50, Vienna, Austria, 2018.

[23] L. Verdoscia and R. Giorgi. A Data-Flow Soft-Core Processor for Accelerating Scientific Calculation on FPGAs. *Mathematical Problems in Engineering*, 2016(1):1–21, 2016.

[24] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the ACM International Conference on Partitioned Global Address Space Programming Models*, page 6, Eugene, USA, 2014.

[25] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R Gao. Using a codelet program execution model for exascale machines: position paper. In *Proceedings of the ACM International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pages 64–69, San Jose California, USA, 2011.

[26] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Targeting distributed systems in fastflow. In *European Conference on Parallel Processing*, pages 47–56. Springer, 2012.

[27] George Matheou and Paraskevas Evripidou. Data-driven concurrency for high performance computing. *ACM Trans. Archit. Code Optim.*, 14(4):53:1–53:26, December 2017.

[28] David E Culler and Gregory M Papadopoulos. The explicit token store. *Journal of Parallel and Distributed Computing*, 10(4):289–308, 1990.

[29] John R Gurd. The manchester dataflow machine. *Computer Physics Communications*, 37(1-3):49–62, 1985.

[30] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion. Hybrid dataflow/von-neumann architectures. *IEEE Trans. on Parallel and Distrib. Systems*, 25(6):1489–1509, June 2014.

[31] R. Giorgi, Z. Popovic, and N. Puzovic. DTA-C: A Decoupled multi-Threaded Architecture for CMP Systems. In *Proceedings of IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 263–270, Gramado, Brasil, 2007.

[32] Theodoropoulos et al. The AXIOM platform for next-generation cyber physical systems. *ELSEVIER Microprocessors and Microsystems*, pages 540–555, 2017.

[33] R. Giorgi. Scalable Embedded Systems: Towards the Convergence of High-Performance and Embedded Computing. In *Proceedings of International Conference on Embedded and Ubiquitous Computing (EUC)*, Porto, Portugal, 2015.

[34] C. Alvarez et al. The AXIOM software layers. In *Proceedings of IEEE Euromicro Conference of Digital System Design (DSD)*, pages 117–124, Madeira, Portugal, 2015.

[35] C. Alvarez et al. The AXIOM Software Layers. *ELSEVIER Microprocessors and Microsystems*, 47, Part B:262–277, 2016.

[36] R. Giorgi, N. Bettin, P. Gai, X. Martorell, and A. Rizzo. *AXIOM: A Flexible Platform for the Smart Home*, pages 57–74. Springer, 2016.

[37] D. Theodoropoulos et al. The AXIOM project (Agile, eXtensible, fast I/O Module). In *Proceedings of International Conference on Embedded Computer Systems: Architecture, MOdeling and Simulation (SAMOS)*, pages 262–269, Samos, Greece, 2015.

[38] Antonio Filgueras, Miquel Vidal, Marc Mateu, Daniel Jiménez-González, Carlos Alvarez, Xavier Martorell, Eduard Ayguadé, Dimitrios Theodoropoulos, Dionisios Pnevmatikatos, Paolo Gai, et al. The AXIOM project: Iot on heterogeneous embedded platforms. *IEEE Design & Test*, 2019.

[39] R. Giorgi, F. Khalili, and M. Procaccini. AXIOM: A Scalable, Efficient and Reconfigurable Embedded Platform. In *Proceedings of IEEE Design, Automation and Test in Europe (DATE)*, pages 1–6, Florence, Italy, 2019.

[40] R. Giorgi, F. Khalili, and M. Procaccini. A Design Space Exploration Tool Set for Future 1K-core High-Performance Computers. In *Proceedings of ACM Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO)*, pages 1–6, Manchester, UK, 2019.

[41] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. Cotson: infrastructure for full system simulation. *ACM SIGOPS Operating Systems Review*, 43(1):52–61, 2009.

[42] Asim YarKhan and Jack Dongarra. Lightweight superscalar task execution in distributed memory. 2014.

[43] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[44] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[45] Peter Kogge. Next-generation supercomputers. *IEEE Spectrum, February*, 2011.

[46] Pieter Bellens, Josep M Perez, Rosa M Badia, and Jesus Labarta. Cellss: a programming model for the cell be architecture. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 86–es, 2006.

[47] Josep M Perez, Rosa M Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *2008 IEEE International Conference on Cluster Computing*, pages 142–151. IEEE, 2008.

[48] Martin C Rinard and Monica S Lam. The design, implementation, and evaluation of jade. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(3):483–545, 1998.

[49] Mihai Budiu, Pedro V Artigas, and Seth Copen Goldstein. Dataflow: A complement to superscalar. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, pages 177–186. IEEE, 2005.

[50] R. Giorgi and A. Scionti. A scalable thread scheduling co-processor based on data-flow principles. *ELSEVIER Future Generation Computer Systems*, 53:100–108, 2015.

[51] R. Giorgi, R. Badia, F. Bodin, A. Cohen, P. Evripidou, P. Faraboschi, B. Fechner, G. Gao, A. Garbade, R. Gayatri, S. Girbal, D. Goodman, B. Khan, S. Koliaï, J. Landwehr, N. Lê Minh, F. Li, M. Lujàn, A. Mendelson, L. Morin, N. Navarro, T. Patejko, A. Pop, P. Trancoso, T. Ungerer, I. Watson, S. Weis, S. Zuckerman, and M. Valero. TERAFLUX: Harnessing dataflow in next generation teradevices. *ELSEVIER Microprocessors and Microsystems*, 38(8, Part B):976–990, 2014.

[52] N. Ho et al. Simulating a Multi-core x86-64 Architecture with Hardware ISA Extension Supporting a Data-Flow Execution Model. In *Proceeding of IEEE International Conference on Artificial Intelligence, Modelling and Simulation (AIMS)*, pages 264–269, Madrid, Spain, 2014.

[53] Wenao Xie, Chun Zhang, Yuanhang Zhang, Chuanbo Hu, Hanjun Jiang, and Zhihua Wang. An energy-efficient fpga-based embedded system for cnn application. In *2018 IEEE International Conference on Electron Devices and Solid State Circuits (EDSSC)*, pages 1–2. IEEE, 2018.

[54] Francois Abel, Jagath Weerasinghe, Christoph Hagleitner, Beat Weiss, and Stephan Paredes. An fpga platform for hyperscalers. In *2017 IEEE 25th Annual Symposium on High-Performance Interconnects (HOTI)*, pages 29–32. IEEE, 2017.

[55] K. Stavrou et al. Programming abstractions and toolchain for dataflow multithreading architectures. In *Proceedings of IEEE International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 107–114, Lisbon, Portugal, 2009.

[56] L. Verdoscia, R. Vaccaro, and R. Giorgi. A Clockless Computing System based on the Static Dataflow Paradigm. In *Proceedings of IEEE International Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM)*, pages 30–37, Edmonton, Canada, 2014.

[57] Roberto Giorgi, Farnam Khalili, and Marco Procaccini. Translating Timing into an Architecture: The Synergy of COTSon and HLS (Domain Expertise—Designing a Computer Architecture via HLS). *International Journal of Reconfigurable Computing*, 2019.

[58] R. Giorgi, F. Khalili, and M. Procaccini. Energy efficiency exploration on the zynq ultrascale+. In *Proceedings of IEEE International Conference on Microelectronics (ICM)*, pages 52–55, Sousse, Tunisia, 2018.

[59] Jack B Dennis. First version of a data flow procedure language. In *Programming Symposium*, pages 362–376. Springer, 1974.

[60] Jack B Dennis and David P Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd annual symposium on Computer architecture*, pages 126–132, 1974.

[61] KAHN Gilles. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[62] Alan L Davis and Robert M Keller. Data flow program graphs. 1982.

[63] SS Thakkar. Selected reprints on dataflow and reduction architectures. 1987.

[64] Christoph Boden, Andrea Spina, Tilmann Rabl, and Volker Markl. Benchmarking data flow systems for scalable machine learning. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, pages 1–10, 2017.

[65] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. Stream-dataflow acceleration. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 416–429. IEEE, 2017.

[66] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters*, 21(02):173–193, 2011.

[67] Christopher Lauderdale, Mark Glines, Jihui Zhao, Alex Spiotta, and Rishi Khan. Swarm: A unified framework for parallel-for, task dataflow, and distributed graph traversal. *ET International Inc., Newark, USA*, 2013.

[68] Samer Arandi, George Michael, Paraskevas Evripidou, and Costas Kyriacou. Combining compile and run-time dependency resolution in data-driven multithreading. In *2011 First Workshop on Data-Flow Execution Models for Extreme Scale Computing*, pages 45–52. IEEE, 2011.

[69] John R. Gurd, Chris C. Kirkham, and Ian Watson. The manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, 1985.

[70] Gregory M Papadopoulos and David E Culler. Monsoon: an explicit token-store architecture. *ACM SIGARCH Computer Architecture News*, 18(2SI):82–91, 1990.

[71] V Gerald Grafe and Jamie E Hoch. The epsilon-2 multiprocessor system. *Journal of Parallel and Distributed Computing*, 10(4):309–318, 1990.

[72] Davor Capalija and Tarek S Abdelrahman. Microarchitecture of a coarse-grain out-of-order superscalar processor. *IEEE Transactions on Parallel and Distributed Systems*, 24(2):392–405, 2012.

[73] Chao Wang, Xi Li, Junneng Zhang, Peng Chen, Yunji Chen, Xuehai Zhou, and Ray CC Cheung. Architecture support for task out-of-order execution in mpsocs. *IEEE Transactions on Computers*, 64(5):1296–1310, 2014.

[74] Abhishek Kumar Jain, Xiangwei Li, Suhaib A Fahmy, and Douglas L Maskell. Adapting the dyser architecture with dsp blocks as an overlay for the xilinx zynq. *ACM SIGARCH Computer Architecture News*, 43(4):28–33, 2016.

[75] Oliver Pell, Oskar Mencer, Kuen Hung Tsoi, and Wayne Luk. Maximum performance computing with dataflow engines. In *High-performance computing using FPGAs*, pages 747–774. Springer, 2013.

[76] Daniel Orozco, Elkin Garcia, Robert Pavel, Jaime Arteaga, and Guang Gao. The design and implementation of tideflow: A dataflow-inspired execution model for parallel loops and task pipelining. *International Journal of Parallel Programming*, 44(2):278–307, 2016.

[77] Samer Arandi, George Matheou, Costas Kyriacou, and Paraskevas Evripidou. Data-driven thread execution on heterogeneous processors. *International Journal of Parallel Programming*, 46(2):198–224, Apr 2018.

[78] George Matheou and Paraskevas Evripidou. Architectural support for data-driven execution. *ACM Trans. Archit. Code Optim.*, 11(4):52:1–52:25, January 2015.

[79] R. Giorgi. Transactional memory on a dataflow architecture for accelerating haskell. *WSEAS Trans. Computers*, 14:546–558, 2015.

[80] M. Solinas, M. Badia, F. Bodin, A. Cohen, P. Evripidou, P. Faraboschi, B. Fechner, G. Gao, A. Garbade, S. Girbal, D. Goodman, B. Khan, S. Koliaï, F. Li, M. Lujàn, A. Mendelson, L. Morin, N. Navarro, A. Pop, P. Trancoso, T. Ungerer, M. Valero, S. Weis, S. Zuckerman, and R. Giorgi. The TERAFLUX project: Exploiting the dataflow paradigm in next generation teradevices. In *Proceedings of IEEE Euromicro Conference on Digital System Design (DSD)*, pages 272–279, Santander, Spain, 2013.

[81] A. Portero, Z. Yu, and R. Giorgi. TERAFLUX: Exploiting Tera-device Computing Challenges. *ELSEVIER*, 7:146–147, 2011.

[82] R. Giorgi. TERAFLUX: Exploiting Dataflow Parallelism in Teradevices. In *Proceedings of ACM Computing Frontiers*, pages 303–304, Cagliari, Italy, 2012.

[83] N. Ho et al. Enhancing an x86_64 Multi-Core Architecture with Data-Flow Execution Support. In *Proceedings of ACM Computing Frontiers*, pages 1–2, Ischia, Italy, 2015.

[84] R. Giorgi. Accelerating haskell on a dataflow architecture: a case study including transactional memory. In *Proc. Int.l Conf. on Computer Eng. and Applications*, pages 91–100, Dubai, UAE, Feb. 2015.

[85] S. Weis, A. Garbade, J. Wolf, B. Fechner, A. Mendelson, R. Giorgi, and T. Ungerer. A fault detection and recovery architecture for a teradevice dataflow system. In *Proc. IEEE Int.l Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM)*, pages 38–44, Oct. 2011.

[86] S. Weis, A. Garbade, B. Fechner, A. Mendelson, R. Giorgi, and T. Ungerer. Architectural support for fault tolerance in a teradevice dataflow system. *Springer Int.l Journal of Parallel Programming*, 44(2):208–232, Apr 2016.

[87] Ali R Hurson and Krishna M Kavi. Dataflow computers: Their history and future. *Wiley Encyclopedia of Computer Science and Engineering*, 2007.

[88] Ben Lee and Ali R Hurson. Dataflow architectures and multithreading. *Computer*, 27(8):27–39, 1994.

[89] Rex Vedder and Dennis Finn. The Hughes data flow multiprocessor: Architecture for efficient signal and data processing. *ACM SIGARCH Computer Architecture News*, 13(3):324–332, 1985.

[90] Jack B Dennis. The varieties of data flow computers. In *Advanced computer architecture*, pages 51–60. 1986.

[91] Alan L Davis. The architecture and system method of DDM1: A recursively structured data driven machine. In *Proceedings of the Symposium on Computer Architecture*, pages 210–215, Palo Alto, CA, USA, 1978.

[92] A Plas, D Comte, O Gelly, and JC Syre. LAU system architecture: A parallel data driven processor based on single assignment. In *Proceedings of the International Conference on Parallel Processing*, pages 293–302, Enslow,Philip H., 1976.

[93] Merrill Cornish. The ti data flow architectures- the power of concurrency for avionics. *Challenge of the'80 s*, pages 19–25, 1979.

[94] Noriyoshi Ito, Masatoshi Sato, Eiji Kuno, and Kazuaki Rokusawa. The architecture and preliminary evaluation results of the experimental parallel inference machine PIM-D. *ACM SIGARCH Computer Architecture News*, 14(2):149–156, 1986.

[95] Vinod Kathail. A multiple processor data flow machine that supports generalized procedures. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 291–302, 1981.

[96] Masasuke Kishi, Hiroshi Yasuhara, and Yasusuke Kawamura. Dddp-a distributed data driven processor. In *Proceedings of the 10th annual international symposium on Computer architecture*, pages 236–242, 1983.

[97] Toshio Shimada, Kei Hiraki, Kenji Nishida, and Satoshi Sekiguchi. Evaluation of a prototype data flow processor of the sigma-1 for scientific computations. *ACM SIGARCH Computer Architecture News*, 14(2):226–234, 1986.

[98] David E Culler and Gregory M Papadopoulos. The explicit token store. *Journal of Parallel and Distributed Computing*, 10(4):289–308, 1990.

[99] J-L Gaudiot and Y-H Wei. Token relabeling in a tagged token data-flow architecture. *IEEE Transactions on Computers*, 38(9):1225–1239, 1989.

[100] William B Ackerman. A structure processing facility for data flow computers. In *Proc. of*, pages 166–172, 1978.

[101] Herbert HJ Hum, Olivier Maquelin, Kevin B Theobald, Xinmin Tian, Xinan Tang, Guang R Gao, Phil Cupryk, Nasser Elmasri, Laurie J Hendren, Alberto Jimenez, et al. A design study of the earth multiprocessor. In *PACT*, volume 95, pages 59–68. Citeseer, 1995.

[102] Arthur H Veen. Dataflow machine architecture. *ACM Computing Surveys (CSUR)*, 18(4):365–396, 1986.

[103] K. M. Kavi, R. Giorgi, and J. Arul. Scheduled dataflow: Execution paradigm, architecture, and performance evaluation. *IEEE Trans. Computers*, 50(8):834–846, Aug. 2001.

[104] Vason P Srini. An architectural comparison of dataflow systems. *Data Flow Computing: Theory and Practice*, page 101, 1992.

[105] Gregory M Papadopoulos and Kenneth R Traub. Multithreading: A revisionist view of dataflow architectures. In *Proceedings of the 18th annual international symposium on Computer architecture*, pages 342–351, 1991.

[106] R. Giorgi. Scalable Embedded Computing through Reconfigurable Hardware: comparing DF-Threads, Cilk, OpenMPI and Jump. *ELSEVIER Microprocessors and Microsystems*, 63, 2018.

[107] Yale N Patt, Wen-mei Hwu, and Michael Shebanow. HPS, a new microarchitecture: rationale and introduction. *ACM SIGMICRO Newsletter*, 16(4):103–108, 1985.

[108] Jurij Silc, Borut Robic, and Theo Ungerer. Asynchrony in parallel computing: From dataflow to multithreading. *Journal of Parallel and Distributed Computing Practices*, 1(1):3–30, 1998.

[109] Jurij Silc, Borut Robic, and Theo Ungerer. *Processor architecture: from dataflow to superscalar and beyond*. Springer Science & Business Media, 2012.

[110] Robert A Iannucci. Toward a dataflow/von neumann hybrid architecture. *ACM SIGARCH Computer Architecture News*, 16(2):131–140, 1988.

[111] Mitsuhisa Sato, Yuetsu Kodama, Shuichi Sakai, Yoshinori Yamaguchi, and Yasuhito Koumura. Thread-based programming for the em-4 hybrid dataflow machine. *ACM SIGARCH Computer Architecture News*, 20(2):146–155, 1992.

[112] R. Giorgi. Memory decoupled architectures and related issues guest editor's introduction. *IEEE TCCA Newsletter*, pages 2–4, Jan. 2001.

[113] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W Keckler, and Charles R Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 422–433, San Diego CA, USA, 2003. IEEE.

[114] Mahim Mishra, Timothy J Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C Goldstein, and Mihai Budiu. Tartan: evaluating spatial computation for whole program execution. *ACM SIGARCH Computer Architecture News*, 34(5):163–174, 2006.

[115] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *Proceedings of International Symposium on High Performance Computer Architecture (HPCA)*, pages 503–514, San Antoni, USA, 2011. IEEE.

[116] Rishiyur S Nikhil and Gregory M Papadopoulos. T: A multithreaded massively parallel architecture. *ACM SIGARCH Computer Architecture News*, 20(2):156–167, 1992.

[117] David E. Culler, Seth Copen Goldstein, Klaus E. Schauser, and Thorsten Voneicken. TAM-a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18(3):347–370, 1993.

[118] Justin Strohschneider and Klaus Waldschmidt. Adarc: A fine grain dataflow architecture with associative communication network. In *Proceedings of Euromicro Conference. System Architecture and Integration*, pages 445–450, Liverpool,UK, 1994. IEEE.

[119] Lucas Roh and Walid A Najjar. Design of storage hierarchy in multithreaded architectures. In *Proceedings of IEEE International Symposium on Microarchitecture*, pages 271–278, Michigan, USA, 1995. IEEE.

[120] Costas Kyriacou, Paraskevas Evripidou, and Pedro Trancoso. Data-driven multithreading using conventional microprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 17(10):1176–1188, 2006.

[121] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 162–173, New York, NY, USA, 2007. ACM.

[122] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Task superscalar: An out-of-order task pipeline. In *Proceedings of IEEE International Symposium on Microarchitecture*, pages 89–100, Atlanta,USA, 2010. IEEE.

[123] Edward A. Ashcroft and William W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, 1977.

[124] Simon F Wail and David Abramson. Can dataflow machines be programmed with an imperative language. *Advanced Topics in Dataflow Computing and Multithreading*, pages 229–265, 1995.

[125] Kung-Song Weng. *Stream-oriented computation in recursive data flow schemas*. Massachusetts Institute of Technology. Project MAC, 1975.

[126] James R McGraw. The val language: Description and analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(1):44–82, 1982.

[127] Rishiyur S Nikhil, PR Fenstermacher, JE Hicks, and RP Johnson. Id world reference manual. *CSG Memo, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA (April 1987)*, 1987.

[128] Jan Hidders, Natalia Kwasnikowska, Jacek Sroka, Jerzy Tyszkiewicz, and Jan Van den Bussche. Dfl: A dataflow language based on petri nets and nested relational calculus. *Information Systems*, 33(3):261–284, 2008.

[129] James L Peterson. Petri nets. *ACM Computing Surveys (CSUR)*, 9(3):223–252, 1977.

[130] Mark A Roth, Herry F Korth, and Abraham Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems (TODS)*, 13(4):389–417, 1988.

[131] Jean-Luc Gaudiot, Tom DeBoni, John Feo, Wim Böhm, Walid Najjar, and Patrick Miller. The sisal project: real world functional programming. In *Compiler optimizations for scalable parallel systems*, pages 45–72. Springer, 2001.

[132] D Comte, G Durrieu, O Gelly, A Plas, and JC Syre. Parallelism, control and synchronization expression in a single assignment language. *ACM SIGPLAN Notices*, 13(1):25–33, 1978.

[133] John R Gurd, John RW Glauert, and Chris C Kirkham. Generation of dataflow graphical object code for the lapse programming language. In *International Conference on Parallel Processing*, pages 155–168. Springer, 1981.

[134] David Patterson. 50 years of computer architecture: From the mainframe cpu to the domain-specific tpu and the open risc-v instruction set. In *Solid-State Circuits Conference-(ISSCC), 2018 IEEE International*, pages 27–31. IEEE, 2018.

[135] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[136] R. Giorgi. Exploring Future Many-Core Architectures: The TERAFLUX Evaluation Framework. In *Advances in Computers*, Advances in Computers, pages 33–72. Elsevier, 2017.

[137] Debootstrap website. https://wiki.debian.org/Debootstrap.

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

[139] Federico Angiolini, Jianjiang Ceng, Rainer Leupers, Federico Ferrari, Cesare Ferri, and Luca Benini. An integrated open framework for heterogeneous mpsoc design space exploration. In *Proceedings of the Design Automation & Test in Europe Conference*, volume 1, pages 1–6. IEEE, 2006.

[140] Rakesh Kumar, Dean M Tullsen, Norman P Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005.

[141] Dimitris Theodoropoulos, Somnath Mazumdar, Eduard Ayguade, Nicola Bettin, Javier Bueno, Sara Ermini, Antonio Filgueras, Daniel Jiménez-González, Carlos Álvarez Martínez, Xavier Martorell, et al. The AXIOM platform for next-generation cyber physical systems. *Microprocessors and Microsystems*, 52:540–555, 2017.

[142] Rob Dimond, Sébastien Racaniere, and Oliver Pell. Accelerating large-scale hpc applications using fpgas. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pages 191–192. IEEE, 2011.

[143] S. Wong, L. Carro, M. Rutzig, D. Motta Matos, R. Giorgi, N. Puzovic, S. Kaxiras, M. Cintra, G. Desoli, P. Gai, S. Mckee, and A. Zaks. *ERA–Embedded Reconfigurable Architectures*, pages 239–259. Springer New York, 2011.

[144] S. Wong, A. Brandon, F. Anjam, R. Seedorf, R. Giorgi, Z. Yu, N. Puzovic, S. A. McKee, Magnus Sjaelander, and Georgios Keramidas. Early Results from ERA – Embedded Reconfigurable Architectures. In *Proceedings of IEEE International Conference on Industrial Informatics (INDIN)*, pages 816–822, Lisbon, Portugal, 2011.

[145] R. Giorgi, Z. Popovic, and N. Puzovic. Implementing Fine/Medium Grained TLP Support in a Many-Core Architecture. In *Proceedings of International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 78–87, Samos, Greece, 2009. Springer.

[146] R. Giorgi, F. Khalili, and M. Procaccini. Analyzing the impact of operating system activity of different linux distributions in a distributed environment. In *Proceedings of IEEE Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 422–429, Pavia, Italy, 2019.

[147] R. Giorgi. Exploring Dataflow-based Thread Level Parallelism in Cyber-physical Systems. In *Proceedings of ACM International Conference on Computing Frontiers*, pages 295–300, Como, Italy, 2016.

[148] Jianwei Chen, Murali Annavaram, and Michel Dubois. Slacksim: a platform for parallel simulations of cmps on cmps. *ACM SIGARCH Computer Architecture News*, 37(2):20–29, 2009.

[149] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[150] Milo MK Martin, Daniel J Sorin, Bradford M Beckmann, Michael R Marty, Min Xu, Alaa R Alameldeen, Kevin E Moore, Mark D Hill, and David A Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[151] Hui Zeng, Matt Yourst, Kanad Ghose, and Dmitry Ponomarev. Mptlsim: a simulator for x86 multicore processors. In *2009 46th ACM/IEEE Design Automation Conference*, pages 226–231. IEEE, 2009.

[152] Emanuele Del Sozzo. *On How to Effectively Target FPGAs from domain specific tools*. PhD Thesis published by the Politecnico di Milano, dipartimento di elettronica, informazione e bioingegneria, 2019.

[153] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36, 2011.

[154] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[155] Christian Pilato and Fabrizio Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–4. IEEE, 2013.

[156] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn, and Eric Martin. Gaut: A high-level synthesis tool for dsp applications. In *High-Level Synthesis*, pages 147–169. Springer, 2008.

[157] Yana Yankova, Georgi Kuzmanov, Koen Bertels, Georgi Gaydadjiev, Yi Lu, and Stamatis Vassiliadis. Dwarv: Delftworkbench automated reconfigurable vhdl generator. In *2007 International Conference on Field Programmable Logic and Applications*, pages 697–701. IEEE, 2007.

[158] Ace cosy. `http://www.ace.nl`. Accessed: 2019-09-30.

[159] Cadence stratus high-level synthesis. `https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html`. Accessed: 2020-12-20.

[160] Intel high level synthesis compiler. `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html`. Accessed: 2020-12-20.

[161] Sdaccel: Enabling hardware-accelerated software. `https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html`. Accessed: 2020-12-20.

[162] Nitin Chugh, Vinay Vasista, Suresh Purini, and Uday Bondhugula. A dsl compiler for accelerating image processing pipelines on fpgas. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 327–338, 2016.

[163] Atieh Lotfi and Rajesh K Gupta. Rehls: resource-aware program transformation workflow for high-level synthesis. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 533–536. IEEE, 2017.

[164] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. Rt-xen: Towards real-time hypervisor scheduling in xen. In *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, pages 39–48. IEEE, 2011.

[165] Antoni Portero, Alberto Scionti, Zhibin Yu, Paolo Faraboschi, Caroline Concatto, Luigi Carro, Arne Garbade, Sebastian Weis, Theo Ungerer, and Roberto Giorgi. Simulating the future kilo-x86-64 core processors and their infrastructure. In *Proceedings of the 45th Annual Simulation Symposium*, pages 1–7, 2012.

[166] Jason E Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[167] Roberto Giorgi and Gianfranco Mariotti. Webrisc-v: A web-based education-oriented risc-v pipeline simulation environment. In *Proceedings of the Workshop on Computer Architecture Education*, pages 1–6, 2019.

[168] Skyler Windh, Xiaoyin Ma, Robert J Halstead, Prerna Budhkar, Zabdiel Luna, Omar Hussaini, and Walid A Najjar. High-level language tools for reconfigurable computing. *Proceedings of the IEEE*, 103(3):390–408, 2015.

[169] Daniel D Gajski, Nikil D Dutt, Allen CH Wu, and Steve YL Lin. *High—Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[170] Xilinx Inc. Xilinx UltraScale Architecture.

[171] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. Badia, E. Ayguade, and J. Labarta. Productive cluster programming with OmpSs. *Euro-Par Parallel Processing*, pages 555–566, 2011.

[172] L. Dagum and R. Menon. Openmp: An industry standard api for shared-memory programming. In *IEEE International Conference on Computational Science and Engineering*, pages 46–55, Jan 1998.

[173] J. J. Costa, T. Cortes, X. Martorell, E. Ayguade, and J. Labarta. Running OpenMP applications efficiently on an everything-shared SDSM. In *Proceedings of IPDPS*, pages 35–42, 2004.

[174] Vasileios Amourgianos Lorentzos. *Efficient network interface design for low cost distributed systems*. Master thesis published from the Technical University of Crete, 2017.

[175] Eva Geisberger and Manfred Broy. *Living in a networked world: Integrated research agenda Cyber-Physical Systems (agendaCPS)*. Herbert Utz Verlag, 2015.

[176] Massimo Banzi. *Getting Started with Arduino*. Make Books - O'Reilly Media, Sebastopol, CA, 2008.

[177] A. Rizzo, G. Burresi, F. Montefoschi, M. Caporali, and R. Giorgi. Making IoT with UDOO. *Interaction Design and Architecture(s)*, 1(30):95–112, 2016.

[178] R. Giorgi, S. Mazumdar, S. Viola, P. Gai, S. Garzarella, B. Morelli, D. Pnevmatikatos, D. Theodoropoulos, C. Alvarez, E. Ayguade, J. Bueno, A. Filgueras, D. Jimenez-Gonzalez, and X. Martorell. Modeling multi-board communication in the AXIOM cyber-physical system. *Ada User Journal*, 37(4):228–235, December 2016.

[179] R. Giorgi, Z. Popovic, and N. Puzovic. Exploiting DMA to enable non-blocking execution in decoupled threaded architecture. In *Proc. IEEE Int.l Symp. on Parallel and Distributed Processing - MTAAP Multi-Threading Architectures and APplications*, pages 2197–2204, Rome, Italy, May 2009. IEEE.

[180] A. Portero et al. Simulating the Future kilo-x86-64 core Processors and their Infrastructure. In *Proceedings of Annual Simulation Symposium (ANSS)*, pages 62–67, Orlando, FL, 2012.

[181] Hercules2020. https://hercules2020.eu.

[182] Lech Józwiak. Embedded computing technology for highly-demanding cyber-physical systems. *IFAC-PapersOnLine*, 48(4):19 – 30, 2015. 13th IFAC and IEEE Conference on Programmable Devices and Embedded Systems.

[183] Ian Grout. *Digital systems design with FPGAs and CPLDs*. Elsevier, 2011.

[184] V. Milutinovic, J. Salom, N. Trifunovic, and R. Giorgi. *Guide to DataFlow Supercomputing Basic Concepts, Case Studies, and a Detailed Example Postscript*, pages 125–125. Springer, Berlin, DE, Apr 2015.

[185] Skyler Windh, Xiaoyin Ma, Robert J Halstead, Prerna Budhkar, Zabdiel Luna, Omar Hussaini, and Walid A Najjar. High-level language tools for reconfigurable computing. *Proceedings of the IEEE*, 103(3):390–408, 2015.

[186] J.A. Clemente, V. Rana, D. Sciuto, I. Beretta, and D. Atienza. A hybrid mapping-scheduling technique for dynamically reconfigurable hardware. In *Field Programmable Logic and Applications (FPL)*, pages 177–180, Sept 2011.

[187] W. Ahmed, M. Shafique, L. Bauer, and J.H. Karlsruhe. Adaptive resource management for simultaneous multitasking in mixed-grained reconfigurable multi-core processors. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2011 Proc. of the 9th Int'l Conference on*, pages 365–374, Oct 2011.

[188] Amin Sahebi and Roberto Giorgi. Gluon, the high-speed inexpensive and easy interconnect solution.

[189] S. Wesner, L. Schubert, R. Badia, A. Rubio, P. Paolucci, and R. Giorgi. Special section on terascale computing. *ELSEVIER Future Generation Computer Systems*, 53:88–89, July 2015.

[190] João M. P. Cardoso, Tiago Carvalho, José Gabriel F. Coutinho, Ricardo Nobre, Razvan Nane, Pedro C. Diniz, Zlatko Petrov, Wayne Luk, and Koen Bertels. Controlling a complete hardware synthesis toolchain with LARA aspects. *Microprocessors and Microsystems - Embedded Hardware Design*, 37(8-C):1073–1089, 2013.

[191] João M. P. Cardoso, José G. F. Coutinho, Tiago Carvalho, Pedro C. Diniz, Zlatko Petrov, Wayne Luk, and Fernando Gonçalves. Performance-driven instrumentation and mapping strategies using the lara aspect-oriented programming approach. *Software: Practice and Experience*, pages n/a–n/a, 2014.

[192] L. Verdoscia, R. Vaccaro, and R. Giorgi. A matrix multiplier case study for an evaluation of a configurable dataflow-machine. In *ACM CF'15 - LP-EMS*, pages 1–6, 2015.

[193] Yao Wu, Long Zheng, Brian Heilig, and Guang R Gao. HAMR: A dataflow-based real-time in-memory cluster computing engine. *The International Journal of High Performance Computing Applications*, 31(5):361–374, 2017.

[194] John Sargeant and Chris C Kirkham. Stored data structures on the manchester dataflow machine. *ACM SIGARCH Computer Architecture News*, 14(2):235–242, 1986.

[195] Roberto Giorgi and Marco Procaccini. Bridging a Data-Flow Execution Model to a Lightweight Programming Model. Dublin, Ireland, 2019. in press.

[196] Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, pages 289–300, 1993.

[197] Lance Hammond, Vicky Wong, Mike Chen, Brian D Carlstrom, John D Davis, Ben Hertzberg, Manohar K Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. *ACM SIGARCH Computer Architecture News*, 32(2):102, 2004.

[198] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.

[199] WAVEDEOM Gitub. https://github.com/wavedrom/wavedrom/blob/master/LICENSE.

[200] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, New York, NY, USA, 2000. Association for Computing Machinery.

[201] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 21–28, New York, NY, USA, 2005. Association for Computing Machinery.

[202] Andreas Merkel and Frank Bellosa. Balancing power consumption in multiprocessor systems. *ACM SIGOPS Operating Systems Review*, 40(4):403–414, 2006.

[203] http://www.ti.com/lit/ds/symlink/ina219.pdf, December 2015.

[204] Max Grossman, Mauricio Breternitz, and Vivek Sarkar. Hadoopcl: Mapreduce on distributed heterogeneous platforms through seamless integration of hadoop and opencl. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1918–1927. IEEE, 2013.