UNIVERSITÀ DEGLI STUDI DI SIENA
Dipartimento di Ingegneria dell'Informazione e
Scienze Matematiche

UNIVERSITÀ
DI SIENA
1240

A Data-Flow Execution Engine for Scalable Embedded Computing

Marco Procaccini

Ph.D. Thesis in
High Performance and Reconfigurable Architectures for Cyber-Physical Systems

Advisor

Prof. Roberto Giorgi

Siena, January 2020

Acknowledgments

I would like to express my deep and sincere gratitude to my research supervisor, Prof. Roberto Giorgi, for giving me the opportunity to do research and provide invaluable guidance throughout this research. He has taught me the methodology to carry out the research and to present the research works as clearly as possible. It was a great privilege and honor to work and study under his guidance. This thesis would not have been written successfully without his continuous guidance and support.

I am extremely grateful to my life partner Alessandra for her love, understanding and continuing support during my entire research work.

My special appreciation to my labmates and friends Stefano Viola and Farnam Khalili for their enthusiasm and support in providing relevant assistance and help to complete this study.

Last but not least, I would like to express my appreciation to my parents for their unfailing emotional and financial support.

Special thank goes to the external reviewers and examination committee for their comments in improving this thesis.

The work presented in this thesis is partly funded by the European Commission on AXIOM H2020 (id. 645496), TERAFLUX (id. 249013), HiPEAC (id. 779656).

Abstract

The end of Dennard scaling [1], Moore's law [2], and the resulting difficulty of increasing clock frequency forced the engineering community to shift to the multi/many-core processors and multi-node systems as an alternative way to improve performance. An increased core number benefits many workloads, but programming limitations still reduce the performance due to not fully exploited parallelism.

From this perspective, new execution models are arising to overcome such limitations to scale up performance. Execution models like Data-Flow can take advantage of the full parallelism, thanks to the possibility of creating many asynchronous threads that can run in parallel. These threads may encapsulate the data to be processed, their dependencies, and, once completed, write their output for other threads. Data-Flow Threads (DF-Threads) is a novel Data-Flow execution model for mapping threads on local or distributed cores transparently to the programmer [3]. This model is capable of being parallelized massively among different cores, and it handles even hundreds of thousands or more Data-Flow threads with their associated data regions.

Further implementation and evaluation of the DF-Threads model (previously proposed by R.Giorgi [3]) is presented in this thesis. The proposed model can be able to exploit the full parallelism of modern heterogeneous embedded architectures (e.g., the AXIOM-Board [4]). The work relies on introducing the "Data-Flow engine" (DF-Engine), which is able to accelerate the

function execution and spawn many asynchronous, data-driven threads across several general-purpose cores of a multi-core/node system. The DF-Engine can be placed either in software or directly implemented at the hardware level by using a heterogeneous architecture (e.g., the AXIOM-Board). The DF-Engine can handle the creation, the thread-dependency, and the locality of many fine-grain threads, leaving the general-purpose core focusing only on the execution of the threads. This implementation is a hybrid Data-Flow - von-Neumann model, which harnesses the parallelism and data synchronization inherently to the Data-Flow, and yet maintains the programmability of the von-Neumann model.
Starting from the DF-Threads execution model, we analyzed the tradeoffs of a minimalistic API to enable an efficient implementation, which can distribute the DF-Threads either locally across the core of a single multi-core system or/and across the remote cores of a cluster.

Implement and evaluate the proposed model directly on a real architecture requires time, resources, and effort. Therefore, the design has been preliminarily evaluated in a simulation framework, and then the validated model has been gradually migrated into a real board in collaboration with my research group.

The simulation framework presented in this thesis is based on the COTSon simulator [5] and on a set of tools named "MY Design Space Exploration" (MYDSE) [6], which has been implemented and adopted by our research group. Then, we explain how the validation phase of the simulation framework has been performed against real architecture like x86_64 and AArch64. Moreover, we analyzed the impact of different Linux distributions on the execution.

Afterward, we explain the workflow adopted to migrate the design of the DF-Threads execution model from the COTSon simulator to a High-Level Synthesis framework (e.g., Xilinx HLS), targeting a heterogeneous architecture such as the AXIOM-Board [4]. We used a driving example that models a two-way associative cache to demonstrate the simplicity and rapidness of our

framework in migrating the design from the COTSon simulator to the HLS framework. The methodology has been adopted in the context of the AXIOM project [7], which helped our research team in reducing the development time from days/weeks to minutes/hours.

In the end, we present the evaluation of the proposed DF-Threads execution model. We are interested in stressing and analyzing the efficiency of the DF-Engine with thousands or more Data-Flow threads. For this goal, we decided to use the Recursive Fibonacci algorithm, which gives us the possibility to generate such a high number of threads easily. Moreover, we want to study the behavior of the execution model with data-intensive applications for evaluating the performance with memory operations and data movements. For this reason, we adopted the Matrix Multiplication benchmark, which is the main computational kernel of widely used applications (e.g., Smart Home Living, Smart Video Surveillance, Artificial Intelligence).

The proposed design has been evaluated against OpenMPI, which is typically adopted for cluster programming, and against CUDA, a parallel programming language for GPUs. DF-Threads achieve better performance-per-core compared to both OpenMPI and CUDA. In particular, OpenMPI exhibits much more Operating System (OS) kernel activity with respect to DF-Threads. This OS activity slows down the OpenMPI performance. If we consider the delivered GFLOPS per core, DF-Threads is also competitive with respect to CUDA.

Contents

List of Figures

List of Tables

Acronyms

ADC Analog to Digital Converter.

AI Artificial Intelligence.

API Application Programming Interface.

ARM Advanced Risc Machine.

ASIC Application Specific Integrated Circuit.

AXI Advanced eXtendable Interface.

AXIOM Agile eXtensible Input Output Module.

BFS Breadth Firs Search.

BIOS Basic Input-Output System.

BSD Berkeley Software Distribution.

CAN Controller Area Network.

CISC Complex Instruction Set Computer.

CPS Cyber Physical Systems.

CPU Central Processing Unit.

DDR Double Data Rate.

DF-Engine Data-Flow Engine.

DF-Threads Data-Flow Threads.

DFG Data-Flow Graph.

DSE Design Space Exploration.

DSM Distributed Shared Memory.

DSP Digital Signal Processor.

DTHS Dataflow Thread Hardware Scheduler.

ERA Embedded Reconfigurable Architectures.

FFT Fast Fourier Transform.

FIFO First-In, First-Out.

FLOPS Floating point Operations per Second.

FP Frame Pointer.

FPGA Field Programmable Gate Array.

GCC Gnu Compiler Collection.

GFLOPS Giga Floating Point Operation Per Second.

GPIO General Purpose Input/output.

GPP General Purpose Processors.

GPU Graphic Processing Unit.

GRQ Global Ready Queue.

HLS High Level Synthesis.

HP Hewlett Packard.

HPC High Performance Computing.

HS Hardware Scheduler.

I/O Input/Output.

IC Instruction Cache.

ID Identification Number.

ILP Instruction Level Parallelism.

ISA Instruction Set Architecture.

KiB Kilo Binary Byte.

LRQ Local Ready Queue.

LRU Least Recently Used.

LUT Look-up Table.

MC Memory Controller.

MEM Main Memory.

MiB Mega Binary Byte.

MM Matrix Multiplication.

MP Multi Processor.

MPI Message Passing Interface.

MPLT Memory Production Logging Tool.

MSB Most Significant Bit.

MW Mega Watt.

MYDSE	My Design Space Exploration.
NIC	Network Interface Card.
OS	Operating System.
pj	Pico Joules.
PL	Programmable Logic.
PS	Processing System.
RAM	Random Access Memory.
RDMA	Remote Direct Memory Access.
RFIB	Recursive Fibonacci.
RISC	Reduce Instruction Set Computer.
ROI	Region of Interest.
ROM	Room Only Memory.
SARC	Scalable ARChitectures.
SBC	Single Board Computer.
SC	Synchronization Count.
SHL	Smart Home Living.
SoC	System on Chip.
SPI	Serial Peripheral Interface.
SVS	Smart Video Surveillance.
TLB	Translation Lookaside Buffer.
TLP	Thread Level Parallelism.

UART	Universal Asynchronous Receiver-Transmitter.
UHPC	Ubiquitous High-Performance Computing.
USB	Universal Serial Bus.
V-bit	Valid-bit.
VC	Virtual Circuit.
WB	Write Back.
WT	Write Through.

Chapter 1

Introduction

The original motivation for doing research in Data-Flow computing was the possibility of exploiting its massive parallelism [8, 9].
The performance scaling of the processors has basically followed the path of designing deeper pipelines, increasing clock rates, and the number of cores in a chip. However, when the single-chip performance is not anymore sufficient, domain-specific architectures (e.g., FPGA, ASIC, GPU) and heterogeneous architectures become interesting solution [10, 11].

Nevertheless, the full parallelism has not yet been completely exploited on such architectures due to the limitation in the execution model and programming model. The computer science and engineering communities spent effort in trying to exploit the offered parallelism of the architectures by defining specific parallel-programming techniques (e.g., Message Passing Interface - MPI, OpenMP, Cilk++), or domain-specific languages and abstractions (e.g., Map-Reduce, CUDA). However, none of these works has come close to enabling widespread use of parallel programming for a wide array of computing problems [12, 13]. It is necessary to move away from coarse-grain parallelism and adopt fine-grained techniques in order to increase the ability of parallel computers to deliver performance scaling and provide high-performance parallel acceleration [14].

One solution for obtaining fine-grain parallelism is the Data-Flow execution model, thanks to the possibility to create many asynchronous threads that can be run in parallel. The Data-Flow execution model is capable of taking advantage of the full parallelism offered by multi-core and multi-node systems [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28] by introducing a new paradigm, which internally represents applications as a direct graph named

Data-Flow graph (DFG).

Applications are represented as a set of nodes of the DFG, where each node may represent an instruction or anything else (according to Data-Flow principles [9]). Directed arcs between nodes of the Data-Flow graph represent the data dependencies between the nodes. Whenever all inputs are available, the node is ready for firing. It stands in contrast to the von-Neumann execution model, in which an instruction is only executed when the program counter reaches it without considering if it can be executed earlier or not. The key advantage is that more than one instruction can be executed at once in a Data-Flow execution model. (indeed, superscalar processors exploit this Data-Flow principle internally through the dynamic scheduling of instructions). Thus, when a node becomes ready, it fires, and its execution results asynchronous. Consequently, ready nodes can potentially be executed in parallel. Past attempts to design an entire machine based on that principle were not successful (e.g., Manchester Data-Flow Machine, Explicit Token Store architecture [29, 30]) mainly due to the excessive fine granularity of the approach (e.g., at the instruction level). One significant drawback of the Data-Flow model was its inability to effectively manage data structures, memory organizations and traditional programming languages [31]. Therefore, in order to increase the performance of multi-core systems, a possible solution can be to design a hybrid architecture that combines the Data-Flow and von-Neumann models of computation [31, 32, 33].

In the context of the AXIOM project [34, 35, 36, 37, 38, 4, 39, 40, 7], where the aim was the definition of a software/hardware architecture suitable for heterogeneous embedded system in the Cyber-Physical domain, a further implementation and validation of the Data-Flow Threads execution model (called DF-Threads) has been developed. The proposed implementation of the DF-Threads allows offloading tasks to a "Data-Flow Engine" (DF-Engine) in order to accelerate the function execution and the distribution of the threads across multiple core/node. The DF-Engine can be implemented both in software or directly designed at the hardware level in a heterogeneous architecture (e.g., AXIOM-Board [4]).

As target heterogeneous architecture, it is decided to rely on the AXIOM-Board, which is one of the essential contributions of the AXIOM project and it is based on FPGA and an embedded processor, i.e., Xilinx Zynq Ultrascale+ [4]. 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 partly into accelerators (on programmable logic) and, at the same time, to distribute the computation workloads across boards.

The directly Design Space Exploration (DSE) and debugging performed on the real architectures may reach many hours of days even with powerful workstations [6, 41]. As such, moving already-validated architectures to the real systems may save significant time, effort and, as a result, facilitates the design development. Basing on these motivations, the first step was to explore the design space by using the Design Space Exploration toolset (named MYDSE) [6] and the COTSon simulator [5] to find the most efficient implementation of the execution model. Finally, the DF-Threads design was migrated on a cluster of AXIOM-Boards (Figure 1.1).

In order to evaluate the performance of the DF-Threads execution model, two benchmarks have been selected: Matrix Multiplication and Recursive Fibonacci. The Matrix Multiplication kernel represents one of the computationally intensive portions of real-life applications like Smart Video Surveillance (SVS) and Smart Home Living (SHL) [38]. These applications are very computational demanding since they require analyzing a huge number of scenes coming from multiple cameras located, e.g., at airports, homes, hotels, shopping malls. The Recursive Fibonacci benchmark can stress and analyze how the proposed hardware scheduler manages a huge number of Data-Flow threads across multiple cores and boards.

This thesis mainly focuses on the simulation framework definition for the proposed Data-Flow execution model, which allows us a more systematic and exhaustive analysis of the performance. This preliminary work required the development of a methodology and tools, which has therefore reduced the time for implementing and evaluating a large set of applications.

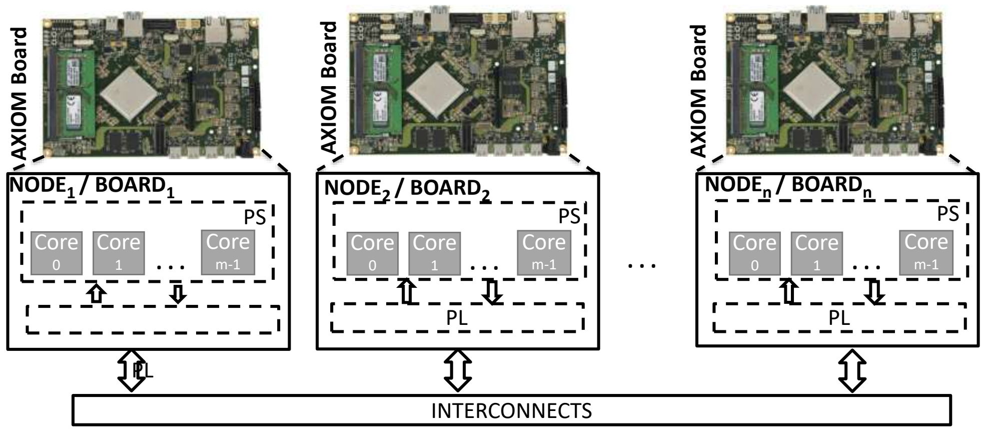


Figure 1.1: *Architecture of the distributed system based on the AXIOM boards. The Distributed System consists of N Nodes based on an SoC, which includes a Processing System (PS) and Programmable Logic (PL). The nodes of the system are connected through USB-C cables without the need for an external switch.*

1.1 Problem Statement

The end of Dennard scaling [1] and Moore's law [2] force the engineering community to switch from a single energy-hogging processor per microprocessor to multiple efficient processors or cores per chip.[10].

However, Amdahl's law demonstrates the diminishing returns from increasing the number of processors due to their limitation of the parallelism by the sequential portion of a task ("strong scalability") [42]. Afterward, Gustafson re-valuated Amdahl's law by considering not only the increase in hardware resources but also the size of the problem ("weak scalability") [43]. Furthermore, performance limitations also come from the lack of exploiting the full parallelism offered by the architectures [13]. As we can see from Figure 1.2, the above mentioned four laws have had a significant impact on the processor performance for the past 40 years. Following this trend, performance on standard processor benchmarks will not double before 2038 [10].
Moreover, the performance projections out to exascale in Figure 1.3 show that only a technological path that has the hope of achieving exascale by 2020 involves a heterogeneous architecture consisting of both lightweight and heavyweight cores [11].

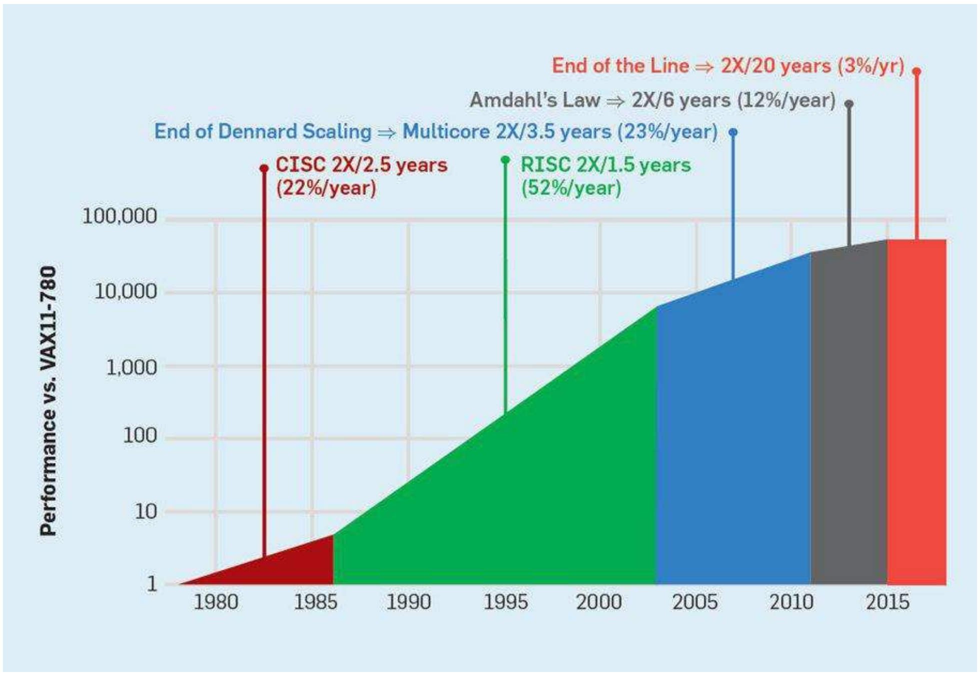


Figure 1.2: *SPECCPUint performance trend per year for 32-bit and 64-bit processor cores over the past 40 years. The figure also shows how the effect of the Dennard scaling, the Moores law and the Amdhal law on the processor performance for the past 40 years [10].*

The Data-Flow execution model is capable of taking advantage of the full parallelism offered by multi-core/multi-node heterogeneous architectures [16, 31, 32, 44]. Each Data-Flow thread is a node of the Data-Flow graph, which only executes when all of its inputs are available. In a Data-Flow based execution, a program could be executed not necessarily following its linear order but in a partial order which relies on the data dependencies. As a result, individual partial orders are data-independent and can be executed in parallel. The length of the independent data path is the expression of the granularity of parallelism. As defined by Dennis [9], there exist multiple instances of architecture that can exploit the potential of the Data-Flow execution model (explicit Data-Flow architectures). Otherwise, they cannot totally replace the conventional general-purpose processors (GPP) due to some limitations of the

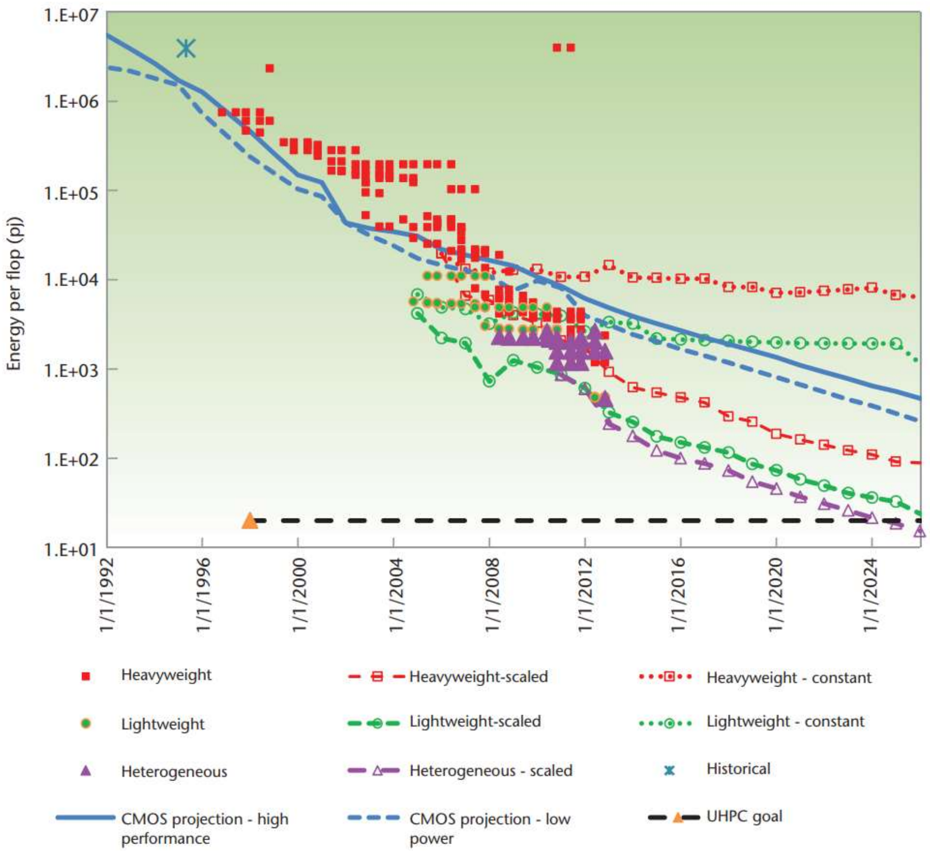


Figure 1.3: *Projections in energy per flop show that only the hybrids have a chance of reaching exaflop performance within the 20-megawatt (MW) power budget by 2024, but with the caveat that they must improve efficiency to offer commensurate improvements in application performance. (Here, pJ =picojoules and UHPC = ubiquitous high-performance computing.) [11].*

execution model. For example, control transfer could be more expensive in some Data-Flow models, and the latency cost of explicit data communication can be prohibitive [45].

In order to exploit the full parallelism offered by multi-core/multi-node, a hybrid model is presented in this work where the threads are scheduled following the DF model, and the instructions are executed in a control-flow

manner. Moreover, a DF-Engine helps the execution in accelerating the function and distributing many asynchronous threads across a multi-core/node system. The DF-Engine can be implemented both in software and in hardware by exploiting the reconfigurability of heterogeneous architecture (e.g., AXIOM-Board). Heterogeneous architecture can be composed by General Purpose Cores (GPP) and Field Programmable Gate Array (FPGA). GPP cores allow us to be suitable for a large set of applications, and FPGAs are known for their reconfigurability and power efficiency. This makes them a suitable choice for being deployed in the many-threads Data-Flow execution models, as well as providing a spatial substrate for mapping DF-threads. These models evolve around the optimization of data mobility and exploiting massive parallelism among thousand of DF-Threads to offer more modularity and higher performance [3, 9, 24, 46, 47, 48, 49].
A scalable Hardware Scheduler (HS), being used in PL, runs in the background, and it is tightly coupled with the GPP cores. The HS is responsible for the data synchronization and DF-Threads distribution among the GPP cores of multiple heterogeneous boards, exploiting the computational power of the GPP cores only for the execution of the threads. The overall architecture has been modeled and tested into the COTSon simulator [5], and then, the model has been mapped and designed for being employed in a heterogeneous architecture (i.e., AXIOM-Board [4]).

1.2 Goals and Challenges

Given these motivations, the thesis focuses on the further implementation and evaluation of a Data-Flow execution model called DF-Threads for heterogeneous embedded systems with the capabilities to fully exploit the parallelism of multi-core and multi-node heterogeneous architectures. The implementation of the proposed Data-Flow execution model required these goals and challenges:

- The validation of the simulation framework against real architecture like x86_64 and AArch64. This goal requires the implementation of a tracing system into the COTSon simulator, which permits us to analyze in detail the executed instructions.

- The extension of the DF-Threads API with the integration of a DF-Engine. The API execution can be accelerated by DF-Engine, which can be implemented either in software or in hardware. Furthermore, the DF-Engine should balance the distribution of the DF-Threads across multiple cores/nodes. The main challenge of this goal is the implementation of a timing model into the COTSon simulator, which can model the execution of the new API and the DF-Engine.

- The performance analysis for the proposed Data-Flow execution model with different benchmarks in a multi-core and multi-node environment. This evaluation requires the utilization of specific tools for the design space exploration, with the ability to extract overhead coming from external sources (e.g., Operating System activity).

1.3 Contribution

These are the contributions of this thesis:

1. The validation of the adopted simulation framework against real architecture like x86_64 and AArch64 [50] (see Chapter 2).

2. The implementation of a tracing system into the simulation framework, which allows us to quickly analyze the overhead of the proposed Data-Flow execution model (see Chapter 2).

3. The extension of the programming interface of the DF-Threads, which is based on a minimalistic API, for the embedded system domain [51] (see Chapter 3).

4. The impact evaluation of several Linux distributions during the execution of the proposed DF-Threads execution model [52] (see Chapter 3).

5. The definition of a DF-Engine for the proposed Data-Flow execution model for accelerating the execution of functions and distributing the Data-Flow Threads across multiple core/node [53, 54] (see Chapter 4).

6. The description of the methodology adopted to translate the preliminary design of the DF-Threads execution model from the COTSon simulator into a heterogeneous architecture like the AXIOM-Board [41] (see Chapter 4).
7. The evaluation of the proposed DF-Threads execution model in comparison with other well-known parallel programming models like OpenMPI and CUDA by using the Recursive Fibonacci and Matrix Multiplication benchmarks [51] (see Chapter 4).

1.4 Publications

- International and National Journals
  1. R. Giorgi, F. Khalili, and 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, London, UK, 3 November 2019, pp. 1-18.
  2. A. Filgueras, M. Vidal, M. Mateu, D. Jimenez Gonzalez, C. Alvarez, X. Martorell, E. Ayguad e, D. Theodoropoulos, D. Pnevmatikatos, P. Gai et al., "The AXIOM project: Iot on heterogeneous embedded platforms", IEEE Design & Test, November 2019, pp 1-6.
- International and National Conference Proceedings
  1. R. Giorgi, M. Procaccini, "Bridging a Data-Flow Execution Model to a Simple Programming Model", IEEE Proceedings of the International Conference on High Performance Computing and Simulation (HPCS), Dublin, Ireland, July 2019, pp. 165-168.
  2. R. Giorgi, F. Khalili, and M. Procaccini, "A design space exploration tool set for future 1k-core high-performance computers", in Proceedings of ACM Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO), January 2019, pp. 16.

3. R. Giorgi, F. Khalili, and M. Procaccini, “Analyzing the impact of operating system activity of different linux distributions in a distributed environment”, in IEEE Proceedings of Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Pavia, Italy, February. 2019, pp. 422-429.
4. R. Giorgi, F. Khalili, and M. Procaccini, “AXIOM: A scalable, efficient and reconfigurable embedded platform”, in IEEE Proceedings of Design, Automation and Test in Europe (DATE), Florence, Italy, March 2019, pp. 16.
5. R. Giorgi, F. Khalili, and M. Procaccini, “Energy efficiency exploration on the zynq ultrascale+”, in IEEE Proceedings of 30th International Conference on Microelectronics (ICM), Sousse, Tunisia, Dec. 2018, pp. 5255.

- Poster Papers

1. M. Procaccini and R. Giorgi, “x86_64 vs AArch64 performance validation with COTSon”, in HiPEAC ACACES-2019, Fiuggi, Italy, July 2019,pp 1-4, abstract.
2. M. Procaccini, F. Khalili, and R. Giorgi, “An fpga-based scalable hardware scheduler for Data-Flow models”, in HiPEAC ACACES-2018, Fiuggi, Italy, July 2018, pp. 14, abstract.
3. F. Khalili, M. Procaccini, and R. Giorgi, “Reconfigurable logic interface architecture for cpu-fpga accelerators”, in HiPEAC ACACES-2018, Fiuggi, Italy, July 2018, pp. 14, abstract.
4. R. Giorgi, F. Khalili, and M. Procaccini, “An fpga-based scalable hardware scheduler for Data-Flow models”, in International workshop on FPGAS for Domain Experts (FPODE), Limassol, Cyprus, Nov. 2018, pp. 11, poster.
5. M. Procaccini and R. Giorgi, “Simulation infrastructure for the next kilo x86_64 chips”, in HiPEAC ACACES-2017, Fiuggi, Italy, Julyy 2017, pp. 9194, abstract.

1.5 Thesis Organization

The work presented in this thesis is organized as follows: in Chapter 2, the Design Space Exploration toolset, the COTSon simulator and the architecture validation against x86_64 and AArch64 architectures is presented; in Chapter 3, the further implementation and the preliminary evaluation of the DF-Threads execution model into the COTSon simulator is described; finally, in Chapter 4, the explanation of the methodology adopted for the translation of the DF-Threads execution model design from the COTSon Simulator to the AXIOM-Board and the evaluation of the implementation against CUDA and OpenMPI is illustrated.

Chapter 2

Design Space Exploration and Automation

Modern embedded systems are increasingly based on heterogeneous Multi-Processor SoC (MP-SoC) architectures [11]. 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 are 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 concerning 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 functional correctness but also may provide accurate timing information.

The DSE tools have been developed by our research team starting from TERAFLUX project [55], in order to evaluate a complex architecture within, e.g., 1000 general-purpose cores and running a full OS like Linux.

DSE tools allow me to model features of architectures that are not yet available on the market in a rapid way.

In this section, the main features of the DSE toolset are presented within the experiments made for validating the simulation environment against real architectures like x86_64 and AArch64 [50].

2.1 The MYDSE toolset

In order to guarantee a proper scientific methodology for experimentation, our research team developed the Design Space and Exploration Tools (DSE Tools) [6, 41], through which it is possible to set up the COTSon simulation environment easily, extract and analyze the results of the experiments Figure 2.1.

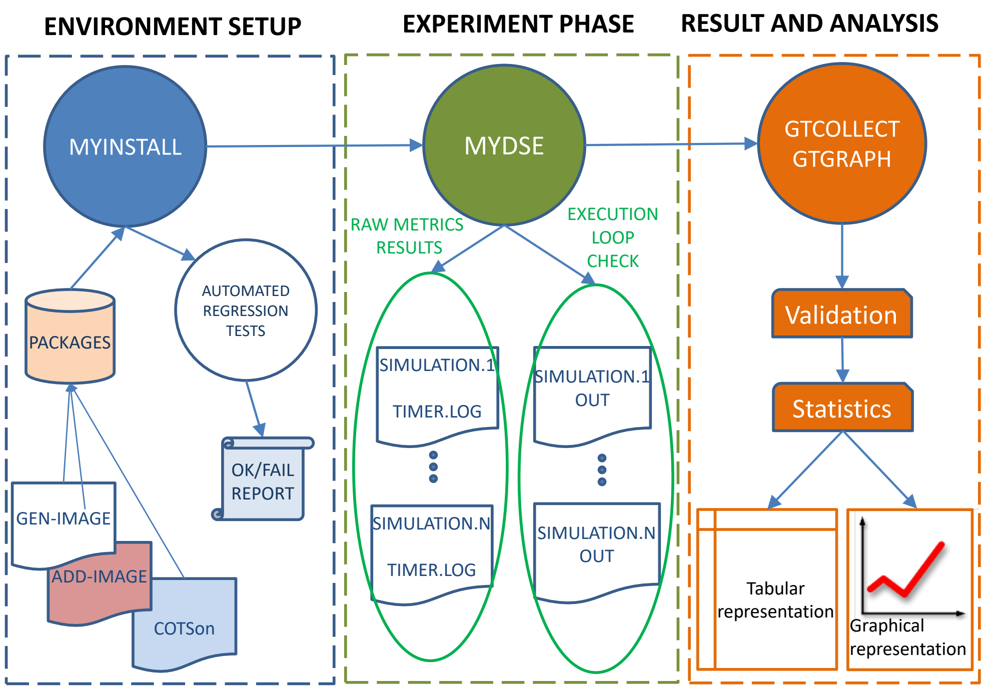


Figure 2.1: *Tool-flow of the Design Space Exploration toolset. The MYINSTALL tool prepares the whole environment and performs automated regression tests in the end. The MYDSE tool takes care of the experiments loop and the re-ordering of the several output files generated by each simulation. Finally, the GTCOLLECTS and GTGRAPH tools collect the results, perform validation and statistical operations on the results, and plot the data in a tabular or graphical format.*

The normal tool flow is to follow the next eight steps.

i) GENIMAGE: to provide an Operating System (OS) for the SimNow virtual machine, the GEN-IMAGE tool has been developed, through which is possible the easy generation of several types of customized OS versions. As depicted in Figure 2.2, the creation of a new hard-drive image (system image) based on a Linux Official public distribution relies on the Debootstrap tool. Furthermore, the GENIMAGE tool generates the hardware description file (BSD file) and the necessary BIOS/ROM for the virtual machine (i.e., AMD BIOS/ROM for SimNow). At the end of the image generation process, the tool performs an automated validation test, using the Tesseract image recognition package to check the screen output.

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 configure a user-based COTSon installation. This tool aims at solving the dependencies needed by the toolset in the host machine, avoiding the manual installation of them. 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 simulation setup without the need for system administrator intervention. The tool runs completely in user-space without the need for root permissions.

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 run into the SimNow machine (see Figure 2.1). Moreover, MYINSTALL allows the choice of the simulation software version, in order to enable more versions of the simulator to co-exist for the regression test.
Finally, the tool performs several regression tests at the end of the installation phase in order to verify if 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: a specific tool 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. MYDSE relies on a small configuration file, named "INFOFILE", which is described in more detail in Section 4.2.
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 versions across the hosts. This allows us to use different Operating Systems,

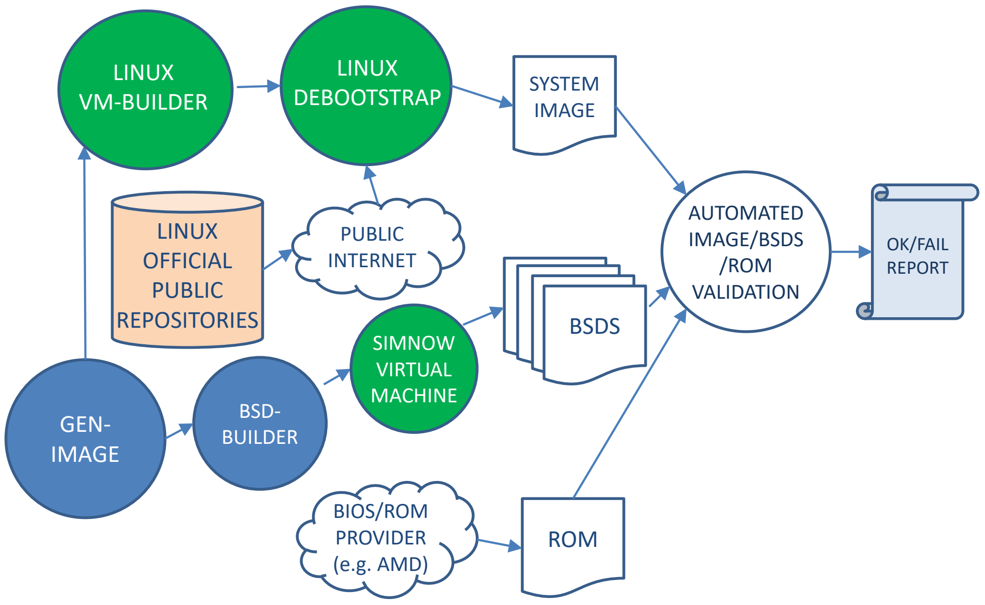


Figure 2.2: *TOOLFLOW for the GENIMAGE tool. The tool generates the hard drive image (system image) and the BSD (hardware description) based on the user requirements. A validation test is made at the end of the workflow, through the Tesserac image recognition package of the screen.*

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 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 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, tasks like collect, analyze, and plot results in a simple way are needed. In this perspective, there is the possibility to 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, additional calculations are performed on the data, e.g., the Coefficient of Variation, to analyze the correctness of the results.
With the GTCOLLECT tool, it is possible to 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.

2.2 Simulation Framework

The proposed simulation framework relies on the HP-Labs COTSon simulation environment [5] and on a set of customized tools, called "MYDSE" that are intended to easily setup the experimental environment, run experiments, extract and analyze the results (see Section 2).

For example, in order to measure the performance of hundreds of multi-core, multiprocessor nodes in an affordable amount of time, we should be able to have a full-system simulator supporting full software stack, memory hierarchy, application benchmarks, and all devices.

The evaluation of the execution model design directly on a real hardware machine is quite challenging and 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 simulator (like COTSon [5]) when it comes to assess and retrieve key metrics of a high-performance computer e.g., 1000 general-purpose cores.

We report in Table 2.1 different approaches like using a physical Cluster, FPGA, and Simulator to evaluate and do research related to the 1000-core computing system (Information revised from the RAMP project [56]).

The main disadvantage of a physical cluster is its high cost and inflexibility towards the modification of architecture as well as poor extensibility in order to reconfigure the Instruction Set Architecture (ISA).

Table 2.1: *Comparison of different approaches for evaluating large computing system. The grade points are scaled between 0 and 5 (grade of 5+ implies the superiority); GPA: Grade Point Average*

	Cluster	FPGA	Simulator
Scalability	5	5)	5
Cost	3	4(0.1-0.2M)	5+(0.01M)
Observability	3	5+	5+
Reproducibility	2	5+	5+
Reconfigurability	3	5+	5+
Credibility	5+	3.5 to 4.5	3
Development time	4	3	5+
Performance (clock)	5(3GHz)	1(GHz)	3
Modifiable	0	4	5
GPA	3.38	3.2 to 3.7	4.8

2.2.1 Simulators Comparison

There are parameters, which make simulators preferable to reach a certain level of performance, scalability, and accuracy, as well as reproducibility and observability. Based on the experience of the previous projects like TERAFLUX [57, 58], ERA [59, 60], AXIOM [61, 4], SARC [62], and according to my research group, we choose to rely on the HP-Labs COTSon simulation infrastructure [5]. In the following, several simulators are compared (see Table 2.2.1), and they are compared with our chosen simulator (i.e., COTSon).

SlackSim [63] is a parallel simulator to model single-core processors. SimpleScalar [64] is a sequential simulator, which supports single-core architectures at the user-level. GEMS [65] is a virtual-machine based full-system multi-core simulator built on top of the Intel Simics virtual-machine. GEMS relies on the timing-first simulation approach, where its timing model drives one single instruction at a time. Even though GEMS provides a complete simulation environment. However, the COTSon simulator provides better performance as we increase the number of modeled cores and nodes. MPTL-sim [66] is a full-system x86_64 multi-core cycle-accurate simulator. In terms of the simulation rate, MPTLsim is significantly faster than GEMS. MPTLsim takes advantage of a real-time hypervisor scheduling technique [67] to build hardware abstractions and fast-forward execution. However, during the execution of the hypervisor, the simulator components like memory, instructions, or I/O are opaque to the user (no statistics is available). On the contrary, for example, COTSon provides an easily configurable and extensible environment to the users [68] with full detailed statistics. Graphite [69] is an open-source distributed parallel simulator that leveraged the PIN package [70] with 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 large core numbers (e.g., 1000 cores) with reasonable speed, but COTSon also provides 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 [5], 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, also includes 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 will be illustrated in Chapter 4.

2.2.2 COTSon Simulator

COTSon simulator [5] 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 data exchange between COTSon and the internal state of SimNow. As can be seen in Figure 2.3, COTSon architecture is made

Table 2.2: *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*	*Decouple-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
MPLTsim	Sequential	Multi-core	no	Not-decoupled - timing first
SlackSim	Parallel	Single-core	No	Not-decoupled - timing first

of three main components:

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, it is possible to customize the x86_64 instruction set of SimNow in order to introduce new instructions for the implementation of the DF-Threads execution model [71].

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

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 time synchronization among all the virtual machines.

Moreover, through simulators like COTSon, it is possible to extend the ISA easily [72, 71], and implement any types of complicated architectures suitable for any execution models like DF-Threads [73, 24, 3], and well-known programming models (i.e., Cilk++, OpenMPI, DSM, Distributed Share Memory) launched on a multi-node architecture. Therefore, COTSon is able to

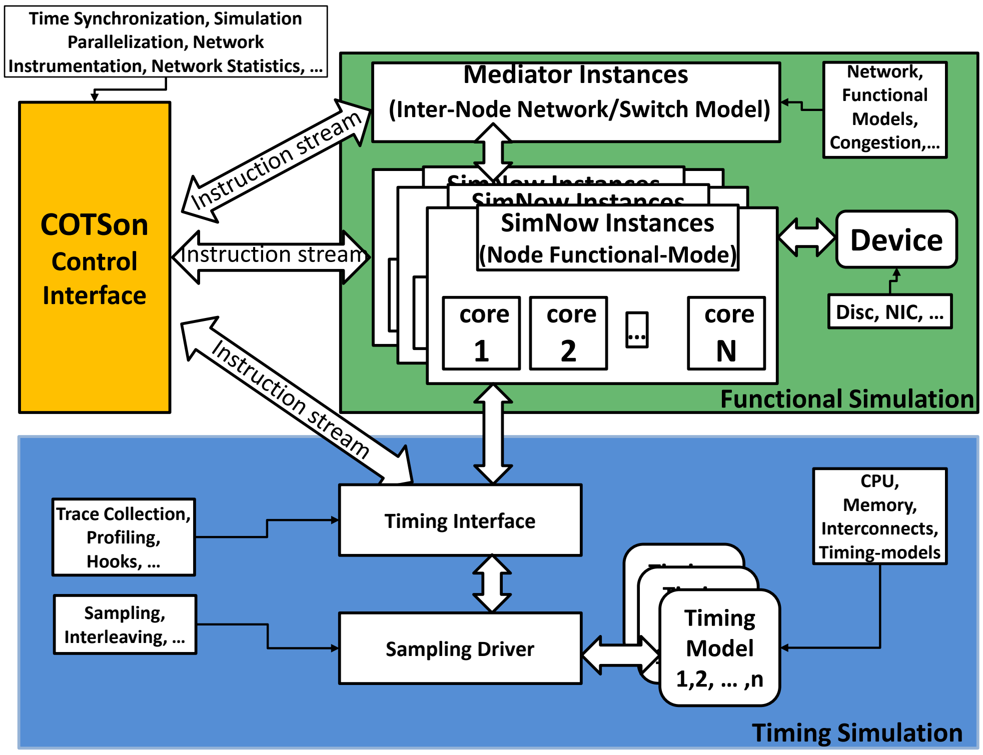

Time Synchronization, Simulation Parallelization, Network Instrumentation, Network Statistics, ...
COTSon Control Interface
Instruction stream
Instruction stream
Instruction stream
Mediator Instances (Inter-Node Network/Switch Model)
Network, Functional Models, Congestion,...
SimNow Instances (Node Functional-Mode)
core 1
core 2
...
core N
Device
Disc, NIC, ...
Functional Simulation
Trace Collection, Profiling, Hooks, ...
Sampling, Interleaving, ...
Timing Interface
Sampling Driver
Timing Model 1,2, ... ,n
CPU, Memory, Interconnects, Timing-models
Timing Simulation

Figure 2.3: *The COTSon simulation framework architecture*

run off-the-shelf Linux distribution, modeling a real situation like interrupts, exceptions, virtual memory management and etc. The simulator gives us the possibility of modeling a complete network infrastructure with several topologies (i.e., star, mesh, torus), allowing us to build a distributed system with many-core/many-nodes (see Figure 2.4).

2.2.3 Performance Validation

The main goal of the performance validation is to verify that the COTSon simulator can be used to evaluate real architectures like x86_64 and AArch64. In the context of the AXIOM project, we use this validation study as a baseline for prototyping the implementation of the proposed Data-Flow execu-

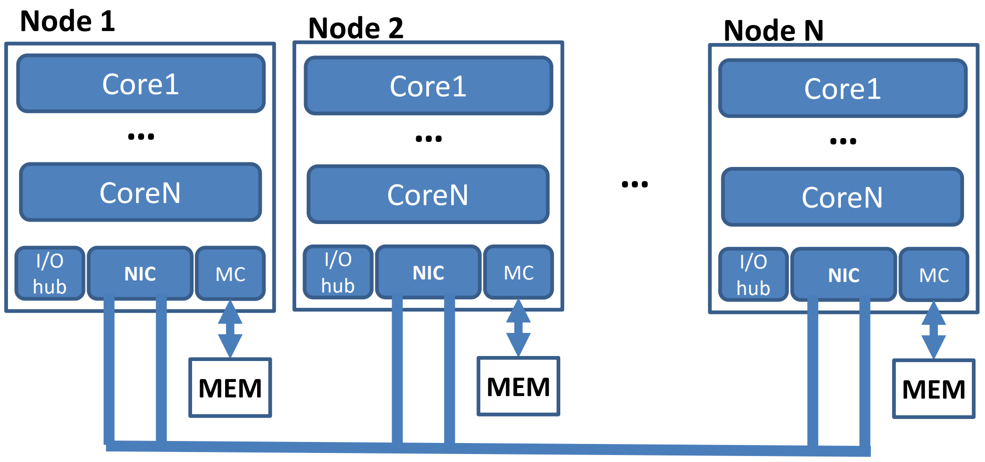


Figure 2.4: *Distributed System architecture with many-cores/many-node made into the COTSon simulator. Each one is modeled as System-on-Chip (SoC). MC=Memory Controller. Network interface controller (NIC) models the network behavior.*

tion model into the COTSon simulator and then migrate the design into the AXIOM-Board. We also wanted to have a confirmation of the findings of previous studies, which discovered that the performance of ARM and x86 does not depend on the specific ISA that is used, but rather it depends on the type and quantity of resources that are included in the platform (e.g., cache, TLB, branch predictor) [74].

The validation task required an extension of the COTSon simulator to analyze in detail the executed instructions. Tracing support has been added to the simulation framework, with the capabilities to extract useful information about the executed instruction (e.g., instruction name, virtual address, physical address, size, latency). The output trace has been compared with the assembly representation of the benchmark executed into the COTSon simulator in order to verify the correct execution flow.

In order to validate the COTSon simulation execution against real architecture like x86_64 and AArch64, we selected four benchmarks: Recursive Fibonacci (RFIB), Matrix Multiplication (MM), Cholesky factorization, and Radix Sort. RFIB is the recursive calculation of the Fibonacci number of a given n. The recursion stops as n becomes smaller than a certain threshold t to avoid too small threads. While not representative of efficient Fibonacci computation, it is still useful because it is a simple test case of a deep tree composed of very fine grain tasks. MM performs a matrix multiplication by a triple nested loop on a dense square matrix, where the matrix is partitioned in sub-matrices or blocks. It represents the computational intensive kernel of many real-life applications (e.g., Smart Video Surveillance, Smart Home Living, and Artificial Intelligence). Cholesky factorization is a useful benchmark for evaluating an efficient numerical solution (e.g., Monte Carlo simulations). The Cholesky algorithm used is the right-looking blocked version. Radix Sort is a well-known sorting algorithm, which avoids comparison by creating and distributing elements into the buckets according to their radix.

Experiments Setup

In order to facilitate the experiments and speed up the Design Space Exploration, we relied on the MYDSE tools presented in Section 2, through which is possible to easily configure the COTSon framework, manage experiments and collect results by reducing the experimentation time from days/week to minute/hours. Thanks to the MYDSE toolset, it is possible to define the complete micro-architecture of the processor by using a higher-level description and also indicate the desired values of the architectural parameters to be explored (see Chapter 4). The tools automatically generate the experiment points for the COTSon simulator, distribute and manages the parallel simulation of the points on as many simulation hosts as possible (e.g., in a cluster system).

Initially, I validated the execution of the COTSon simulator against the Intel i7700 Core [75], by using the architecture parameters reported in Table 2.3. About the AArch64 architecture, we relied on the AXIOM-Board hardware specification, which is a single computer board developed during the AXIOM-Project at the beginning of 2017 [61, 35, 36, 38, 4].

To analyze the sensitivity to the input, the weak scaling of the execution time is considered: the input size is chosen in such a way that any successive value generates a double number of operations. For example, in the MM benchmark, the number of operations varies as $O(N^3)$, where n is the size of the square matrix. Therefore, the size of the matrix has been increased by a factor of $\sqrt[3]{2}$.

Table 2.3: *Micro-Architecture Configurations*

Parameter	COTSon	x86_64	Aarch64
Core	3 GHz, in-order super-scalar	Core Intel i7700 Kaby lake at 4.2 GHz out-of-order	Cortex A53 ARMv8-A at 1.5 GHz, in-order superscalar
Branch Predictor	two-level (history length=18bits, pattern-history table=16 KiB, 17-cycle miss prediction penalty)	two-level (history length=18bits, 17-cycles miss prediction penalty)	global (3072-entries pattern history table) 20-cycles miss prediction penalty
L1 Cache	Private I-cache 32KiB, private D-cache 32KiB, 8-ways, 4-cycles latency	Private I-cache 32KiB 8-ways, private D-cache 32KiB 8-ways 4-cycles latency	Private I-cache 32KiB 2-way, private D-cache 32KiB 4-ways 3-cycles latency
L2 Cache	Private D-Cache 256KiB, 4-ways, 10-cycles latency	Private D-cache 256KiB 4-ways 10-cycles latency	Shared D-cache 1MiB 16-ways 15-cycles latency
L3 Cache	Shared 8MiB, 16-ways, 35-cycles latency	Shared D-cache 8MiB 16-ways 35-cycles latency	no
I-L1-TLB	4KiB pages, 64 entries, 4-ways 1-cycle latency	4KiB pages, 64 entries, 4-ways 1-cycle latency	4KiB pages, 10 entries fully-associative 2-cycles latency
D-L1-TLB	4KiB pages, 64 entries, 8-ways, 1-cycle latency	4KiB pages, 64 entries, 8-ways 1-cycle latency	4KiB pages, 10 entries fully-associative 2-cycles latency
L2-TLB	2MiB, 32 entries, 2-ways	2MiB, 32 entries 4-ways	4KiB, 512 entries, 4-ways

Validation Results

The validation of the results is based on the comparison of the execution time of the sequential execution of the MM, RFIB Cholesky and Radix Sort benchmarks on the COTSon simulator with the two target architectures: Intel i7700 for the x86_64 architecture and ARM Cortex A53 for the AArch64. As depicted in Figure 2.5 and Figure 2.6, the COTSon simulation results follow quite closely the weak scaling that is performed on the Intel i7700 and on the ARM Cortex A53 architectures, for all the benchmarks, and for almost all

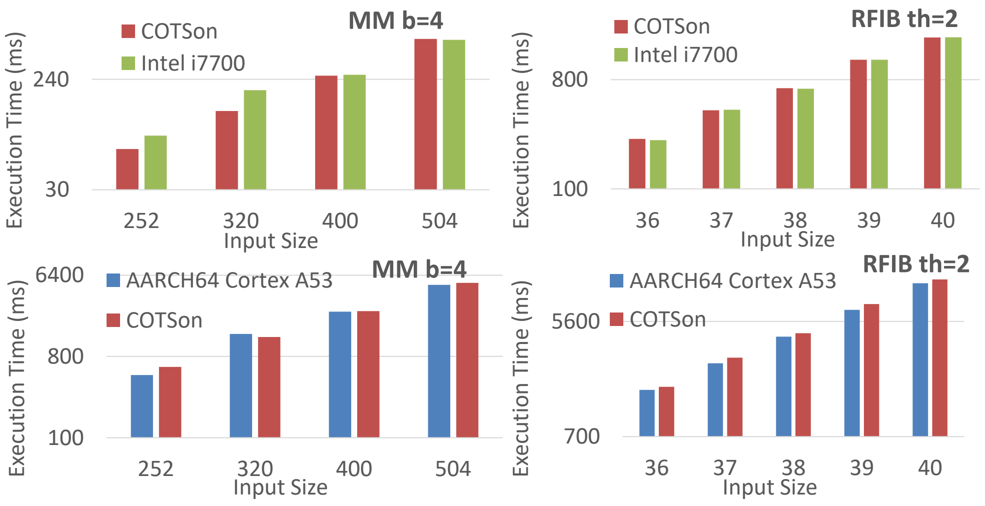


Figure 2.5: *Performance validation of the benchmarks Recursive Fibonacci (RFIB) with 2 as threshold (th) and Matrix Multiply with 4 as block size (b) between the COTSon simulator, the Intel i7700 (top) and the ARM Cortex A53 (bottom) architectures.*

the input range, once an "effective frequency" correction is done. The Matrix Multiplication benchmark shows almost similar results when we increase the input size. At the same time, we can notice differences in smaller inputs due to the variance with a short execution time. Moreover, we tried different block sizes (2,4,8,16,32) without notice of significant variation in the results. The comparison with Radix Sort, and Cholesky is quite similar between COTSon and both real architectures, especially with the AArch64. The Radix Sort benchmark shows a bit more variation due to the short execution time, even with arrays of thousands of elements. The COTSon simulator obtains very close results by using the Recursive Fibonacci algorithm for any input size and with both target architectures.

2.3 Final Remarks

In this Chapter, a set of tools for the Design Space Exploration has been described, based on the COTSon simulator framework, with the aim of supporting a large set of experiments of a multi-node multi-core platform with full OS

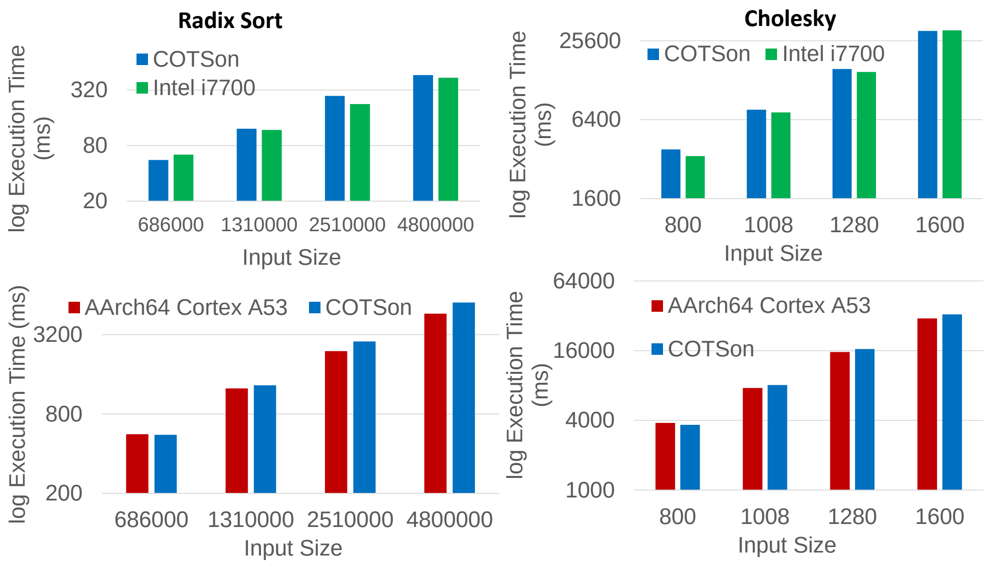


Figure 2.6: *Performance validation of the Radix Sort (left) and Cholesky decomposition (right) between the COTSon simulator, the Intel i7700 (top) and the ARM Cortex A53 (bottom) architectures.*

execution (e.g., 1000 general-purpose processors and real OS activity). The DSE toolset has been implemented by our research group, and they help me in easily set up the simulation environment in less than ten minutes, including several regression tests, saving hours in comparison with manual installation. I was able to 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.
The validation results performed in this thesis on the COTSon simulator against x86_64 and AArch64 architectures permitted to derive information early in the design process.

It has been illustrated how the COTSon simulation framework is capable of simulating both x86_64 and AArch64 architectures and obtain results very close to the real hardware by using the MM, RFIB, Cholesky factorization, and Radix Sort benchmarks. Thanks to this validation, the COTSon simulator gives the possibility of analyzing in depth the behavior of such architectures during the execution of applications along with the related Operating

System activity. Currently, we are working on expanding the benchmark set (e.g., FFT, BFS) with the aim of having a more consolidated validation of our initial findings.

Chapter 3

Data-Flow Execution Engine

Thanks to the continued evolution of the processor design, more and more cores are combined in a chip, and when the single-chip performance is not anymore sufficient, a distributed heterogeneous architecture is a potentially interesting solution [11, 13].

Models like Data-Flow allow us to create many asynchronous threads that can be run in parallel. These threads may encapsulate the data to be processed, their dependencies, and, once completed, write their output for other threads. Data-Flow Threads (DF-Threads) can provide a complete decoupling between execution and memory accesses [3, 76].

Several problems are still open today, such as dynamic task management, find the right balance between the number of threads, runtime overhead, and scalability [77, 78, 79, 80, 81]. The DF-Threads execution model is based on the idea of managing threads dynamically and asynchronously, using a few simple low-level key functions. For each thread, there is a preliminary phase in which its dependencies (inputs) and outputs are specified. After that, all pertinent execution data is encapsulated in a data structure called the “frame”. The scheduler will deal with how, where, and when to run the thread associated with the data (frame).

3.1 Related Work

The Data-Flow (DF) computing model represents a radical alternative to the von-Neumann model, by offering many opportunities for parallel processing. In the beginning, Karp and Miller [82] defined the principles of the Data-Flow with a graph-theoretic model for the description and analysis of parallel computation. After that, the first Data-Flow execution model was developed by Dennis [9], which originally applied the Data-Flow idea to the computer architecture design.

The real implementation of the DF model can be classified into two categories, static and dynamic architectures. The static version is simple, but since the graph is static, every operation can be instantiated only once, and thus loop iterations and subprogram invocations cannot progress in parallel. Some machines were designed based on this model, as an example, the MIT Data-Flow architecture [83, 84], DDM1 [85], LAU [86], and HDFM [87]. The dynamic model tries to overcome some of the restrictions of static DF by supporting the execution of multiple instances of the same instruction template, thereby supporting parallel invocation of loop iterations and subprograms. Notable examples of this model are the Manchester Data-Flow [88] and PIM-D [89].

The Data-Flow model may represent a viable solution for taking advantage of the inherent parallelism available in the applications. However, past implementations of this model have failed to deliver the promised performance due to inefficiencies and limitations. Static Data-Flow is unable to effectively uncover a large amount of parallelism in typical programs (e.g., loops). On the other side, the cost needed for managing the loops in terms of latency, silicon area, and power consumption limits the dynamic Data-Flow [31]. Another important aspect to consider is the notoriously difficult to program the DF architectures because they need the usage of specialized programming languages (e.g., functional languages). Specific Data-Flow languages are necessary to produce large DFG in a way to show as much parallelism as possible for the underlying architecture. Nevertheless, these languages have no notion of explicit computation state, which limits the ability to manage data structures (e.g., arrays). These programmability issues limit the usefulness of the DF machines. Moreover, the lack of a total order on instruction execution makes it difficult to enforce the memory ordering that imperative languages require (e.g., C/C++).

The inherent limitations of both Data-Flow and von-Neumann execution models inspire the creation of a convergent model that can take advantage of synergies to maximize the advantages of both models. The hybrid models, therefore, attempt to harness the parallelism and data synchronization inherent in DF manner. On the other hand, the programming methodology and abstraction are mostly based on von-Neumann models. Hybrid models alleviate DF inefficiencies, either by increasing the basic operating granularity or by limiting the size of the DFG. They integrate abstractions of the control flow and shared data structures. As a result, various hybrid systems use a combinational approach of control flow and DF instruction scheduling techniques. Furthermore, in the hybrid models, nodes of a DFG vary between a single instruction (fine-grain) to a set of instructions (coarse-grain). A further important advantage of hybrid models is noticeable in their memory models. Hybrid models integrate single assignment semantics, inherent to DF, in combination with consistent memory models of the control-flow execution. This relieves one of the main constraints of the pure DF programming: the inability to support shared state and, more precisely, shared data structures. Therefore, hybrid versions are capable of handling imperative languages. As a result, the combination of Data-Flow and von-Neumann models facilitates the design of efficient architectures that benefit from both computing models. The proposed Data-Flow execution model in this thesis relies on the definition of the hybrid model, where the applications are mapped into a DFG and afterward are executed on a control-flow machine.

There are different type of hybrid models classifiable according to the execution semantics and the granularity of the number of operations, from a single instruction to a block of instructions. Out-of-Order model [90] are an extension of the superscalar processors and they utilize the DF model only in the issue and dispatch stages to enhance ILP (e.g., local Data-FLow or micro Data-Flow architecture [91, 92]). Such models can exploit more ILP than the strict von-Neumann architectures, but the block of executable instructions is limited by the technology, and the amount of parallelism they can expose is typically limited. TRIPS [93] TARTAN [94] and DySER [95] execute the instructions within a block in a Data-Flow manner, whereas the blocks are scheduled in control-flow way. This implementation has shown great potential in both performance and power consumption, but the achievable ILP is still limited by a few instructions in the block size. On the contrary, the proposed DF-Threads implementation schedules the blocks by following the Data-Flow model, and the instructions included in the blocks are executed in a control-flow manner. Further examples of similar implementation are Star-T (*T) [96], TAM [97], ADARC [98], EARTH [99], MT. Monsoon [100], Plebbes [101], SDF [102], DDM [103], Carbon [104], and Task Superscalar [105]. This strategy has taken advantage of the recent growth in the number of parallel hardware structures like cores and FPGA. They are able to exploit all these resources more effectively than conventional (e.g., von-Neumann) models while keeping the programming model inside the blocks, which facilitates the programmability of such architectures.

3.2 A Lightweight Data-Flow Programming Model

The DF-Threads *execution model* relies on the Data-Flow graph, in which each node of the graph represents a fine-grain thread named DF-Thread. The execution of the DF-Threads follows the producer-consumer paradigm, in which a DF-Thread (consumer) can execute only when all its inputs have been produced by other DF-Threads (producers). The lifetime of a DF-Thread is defined by 4 API calls (potentially instructions) [55], which are briefly recalled in Table 3.2.

For the purpose of easier mapping generic program code, the API has been elevated to a lightweight programming model, in which DF-Threads are simple C functions without the need to use the stack for passing parameters and with the addition of Data-Flow semantics [51]. A DF-Thread can, therefore, be implemented, as illustrated in Figure 3.1, with explicit management of the input frame (where input data is stored) in coordination with the linker. In Table 3.2, it is represented the semantics of the *df_ldframe*, and *df_destroy* typically placed respectively at the beginning and end of the DF-Thread. The possible transformation of the Recursive Fibonacci and Matrix Multiplication code into a DF-Threads is shown in Figure 3.2 and Figure 3.3.

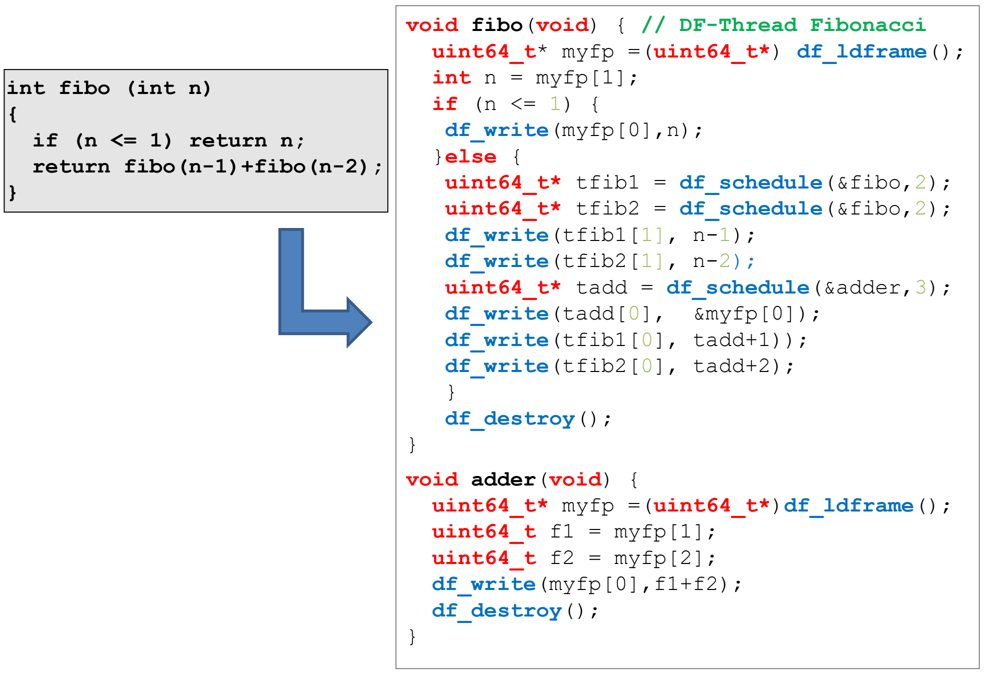
```
void a_df_thread (void) {
    df_ldframe()
    <df_thread_body>
    df_destroy()
}
```

Figure 3.1: *A DF-Thread in C language.*

In this case, the original code (left) is mapped into two DF-Threads named *fibo* and *adder* (right). The *df_schedule* defines how many inputs the next instances will receive. The *df_write* fills up the input frames of the next instances. As soon as all inputs of the target thread have been written, the target thread is executable. In the end, the DF-Thread notifies that its metadata can be removed (*df_destroy*). This approach allows us to use a standard compiler (i.e., GCC) for producing the binary for the target architecture (e.g., x86_64, AArch64).

Table 3.1: *DF-Threads API*

void* df_ldframe();
Loads the data from the (self) input frame
void* df_schedule(void* ip, uint64_t sc);
This function allocates the resources: a data frame of size sc words and returns its Frame Pointer (fp); ip specifies the Instruction Pointer to the first instruction of the code. The allocated DF-Thread is not executed until its sc reaches 0.
void df_write(void* fp, uint64_t val);
The data val is stored into the frame pointed by fp. Note: it is assumed that writes are snooped by the architecture (in particular by the hardware scheduler) so that, for every word that is written, the sc of the DF-Thread - to which the fp belongs is decremented.
void df_destroy();
The thread that invokes df_destroy finishes and its frame is freed.



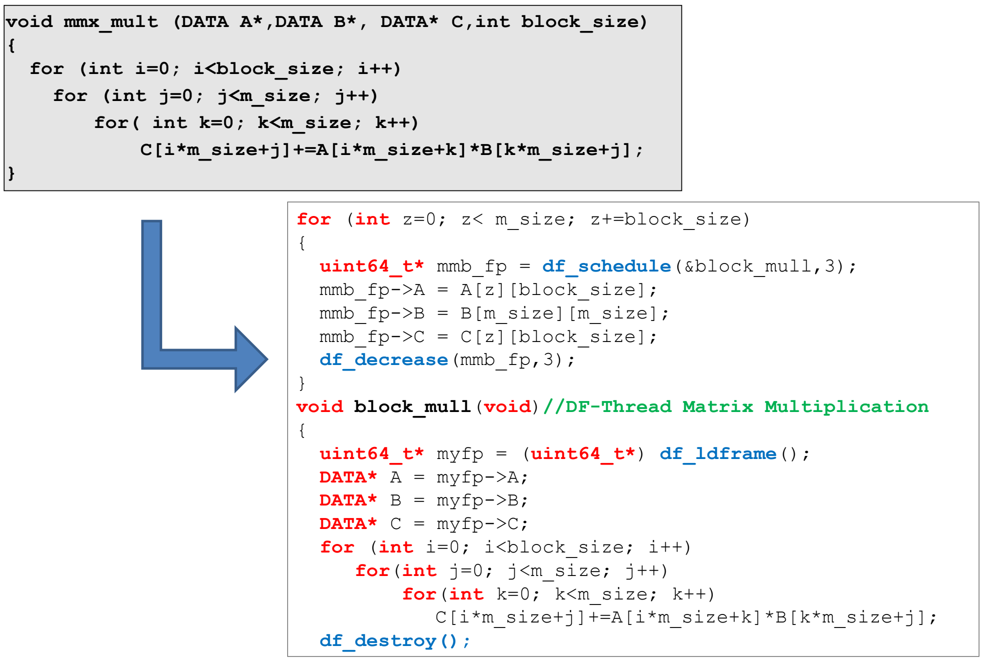
```
int fibo (int n)
{
  if (n <= 1) return n;
  return fibo(n-1)+fibo(n-2);
}
```

```
void fibo(void) { // DF-Thread Fibonacci
  uint64_t* myfp =(uint64_t*) df_ldframe();
  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_destroy();
}
void adder(void) {
  uint64_t* myfp =(uint64_t*)df_ldframe();
  uint64_t f1 = myfp[1];
  uint64_t f2 = myfp[2];
  df_write(myfp[0],f1+f2);
  df_destroy();
}
```

Figure 3.2: *Example of translation of the Recursive Fibonacci program from C code into DF-Threads API. The two main operations (fibo and adder) are translated into the DF-Threads API.*

3.3 Methodology

In this section, the experimental workflow used during the further implementation and evaluation of the DF-Threads execution model are described. In a preliminary phase, the proposed DF-Threads execution model has been implemented into a customized version of the HP Labs COTSon Simulator [71, 106, 5, 68, 53], in order to find the best architecture setup for a heterogeneous architecture (e.g., the AXIOM-Board). The simulators might not show satisfying credibility, but as they evolve, their credibility also improves (see validation study in Chapter 2). The main problem of simulators is their less performance in comparison with a physical cluster. However, the simulators are very useful for approaching a reasonable level of accuracy, scalability, and simulation speed. Importantly, COTSon [5] offers a flexible simulation environment, which made it possible to design our DSE, and add new instruc-



```
void mmx_mult (DATA A*,DATA B*, DATA* C,int block_size)
{
  for (int i=0; i<block_size; i++)
    for (int j=0; j<m_size; j++)
       for( int k=0; k<m_size; k++)
           C[i*m_size+j]+=A[i*m_size+k]*B[k*m_size+j];
}
```

```
for (int z=0; z< m_size; z+=block_size)
{
  uint64_t* mmb_fp = df_schedule(&block_mull,3);
  mmb_fp->A = A[z][block_size];
  mmb_fp->B = B[m_size][m_size];
  mmb_fp->C = C[z][block_size];
  df_decrease(mmb_fp,3);
}
void block_mull(void)//DF-Thread Matrix Multiplication
{
  uint64_t* myfp = (uint64_t*) df_ldframe();
  DATA* A = myfp->A;
  DATA* B = myfp->B;
  DATA* C = myfp->C;
  for (int i=0; i<block_size; i++)
     for(int j=0; j<m_size; j++)
         for(int k=0; k<m_size; k++)
            C[i*m_size+j]+=A[i*m_size+k]*B[k*m_size+j];
  df_destroy();
```

Figure 3.3: *Example of translation for a blocked version of the Matrix Multiplication program from C code into DF-Threads API. The initial for loop divides the matrix into multiple asynchronous DF-Thread. Each DF-Thread executes the classical triple nested loop on the portion of the matrix assigned. The variable "m_size specifies the size of the matrix.*

tions [71, 72, 68] in order to evaluate a multi-core architecture with a dataflow execution model like DF-Threads.

In our research group, we use COTSon to offer a flexible DSE toolset that can easily adopt new hardware/software platforms and support scalability for a multi-node architecture. For instance, to address the challenges of a 1k-core architecture, it should be necessary to have a full-system simulation including Operating System (OS), application benchmarks, a memory hierarchy, and all peripherals as well.

3.4 Operating System Impact

Nowadays, the size of high-performance computers (HPCs) is growing to thousands of nodes, and almost all top supercomputers deploy Linux on each node. In order to achieve concurrency with the asynchronous parallelization of jobs, an operating system (OS) may be responsible for distributing the parallel tasks across the available hardware resources. In that sense, mechanisms like inter-process communication, data synchronization, and thread/process management are all parts of OS responsibility [107]. Hence, part of the system performance is consumed by OS to schedule and manage the hardware resources. Meanwhile, system performance begins to be degraded in large-scale applications using hundreds to thousands of nodes due to the time gaps between speedy processes and lagging ones [108, 109, 110]. This degradation arises from the variability due to OS impact for each node system activity.

In Design Space Exploration (DSE), understanding the OS impact on system performance while running a certain application is crucial. As [108, 109] discovered, the performance of a large-scale application running OS begins to drop-down as the number of nodes increases. However, in order to check the OS intervention, specifically in scaling up the cluster, a complete simulation ranging from multi-core nodes up to full clusters with complete network simulation is unconditionally needed. COTSon is relatively mature and can support scalability and running OS, which permits us to extract kernel activities while running a multi-node application.

As described in section 2.2, COTSon uses a functional approach for the simulation, giving us the possibility to model the system features through the specification of the timing models. A timing model has been defined for a CPU x86_64 instruction set based on extracting the kernel activity, where the kernel references are extracted by analyzing the instruction address (in case of fetch) and the effective address of the load/store instructions. As depicted in Figure 3.4, the methodology to identify the kernel references consists of analyzing the most significant bit of memory address accessed by the current reference, and if the bit is equal to 1, it means that the reference is trying to read or write in the Kernel space memory section. Consequently, the reference is marked as a kernel type. Finally, this methodology has been integrated into

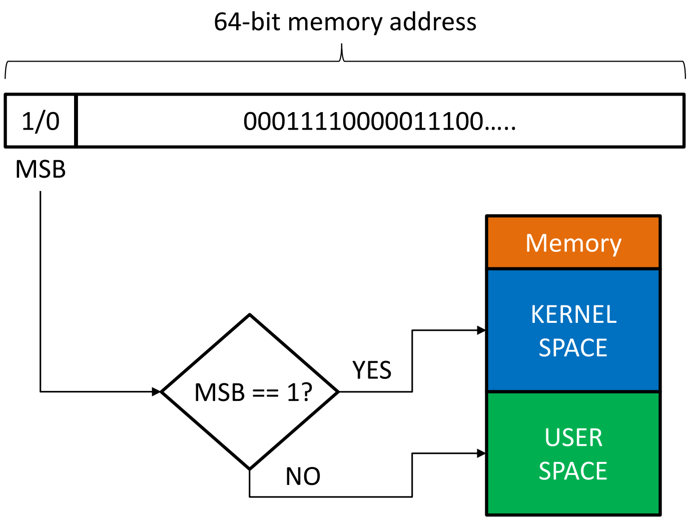


Figure 3.4: *Kernel reference identification technique in COTSon simulator. The Most Significant Bit (MSB) of the current instruction is analyzed and if the MSB is equal to 1, it means that the instruction is a Kernel Instruction.*

the CPU timing model of COTSon [52].

3.5 Evaluation results

In this section, the evaluation performed into the COTSon simulator on the proposed Data-Flow execution engine by using the Matrix Multiplication benchmark is presented.

We analyzed how the execution time differs by varying the Operating System. Moreover, we studied key aspects (i.e., L2 Miss Rate, 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 is a square matrix of 512 as size with 8 as block size. The COTSon simulator architecture is configured with different L2 caches sizes for studying the impact of the OS on the cache hierarchy. Also, we decided to use a single core configuration to avoid possible migrations of OS activity between core. The microarchitecture configurations used in the design space exploration are summarized in Table 3.2.

The OS distributions that have been used in the following experiments are based on the Linux distribution Ubuntu like. The daemons and services did not disable for evaluating each Linux distribution as they were released in the official repository (off-the-shelf). Four different kernel versions were selected to perform our tests:

- karmic64: it is the Ubuntu 9.10 LTS version with the kernel version 2.6.31-22. This distribution focuses on improvements in cloud computing as well as further improvements in boot speed.
- tfx4: it represents the 10.10 Maverick release of the Ubuntu distribution with the kernel version 2.6.35-28.
- trusty-axmv3: it is the Ubuntu 14.04 LTS with Kernel version 3.3.13.0-32. The main improvements were based on increasing performance and maintainability.
- xenv0: it represents the 16.04 Xenial LTS release of the Ubuntu distribution with kernel version: 4.4.0-31.

This simulation framework permits the easy exploration of our benchmark execution while we vary, e.g., the number of nodes (1, 2, 4) and the OS. The input size of the shown example is fixed (matrix size=512 elements).

As can be seen in Figure 3.7, there is a large variation of the kernel cycles between xenv0 (Linux Ubuntu 16.04) and the other three Linux distributions. It indicates us to put attention to the precise configuration of many daemons that run in the background, and that may affect the activity of the system. While doing tests directly on the FPGA, it would not have been easy to understand that the OS activity absorbed most of the time taken by the execution: a designer could have taken it for granted, or he/she could not even have the possibility of changing the OS distribution for testing the differences, since the whole FPGA workflow is typically oriented to a fixed decision for the OS (e.g., Xilinx Petalinux). The situation is even worse for cache parameters or for the number of cores, since the designer might be forced to choose a specific configuration.

Table 3.2: *COTSon microarchitectural parameters to model our target machine.*

Parameter	Description
SoC	1-core connected by a shared-bus, IO-bus, MC, high-speed transceivers
Core	3GHz, in-order super-scalar
Branch Predictor	Two-level (history length=14bits, pattern-history Table=16KiB, 8-cycle miss-prediction penalty)
L1 Cache	Private I-cache 32KiB, private D-cache 32 KiB, 2 ways, 3-cycle latency
L2 Cache	Private 64,256,1024 KiB, 4 ways, 5-cycle latency
L3 Cache	Shared 4MiB, 4 ways, 20-cycle latency
Coherence protocol	MOESI
Main Memory	1 GiB, 100 cycles latency
I-L1-TLB, D-L1-TLB	64 entries, direct-access, 1-cycle latency
L2-TLB	512 entries, direct access, 1-cycle latency
Write/Read queues	200 Bytes each, 1-cycle latency

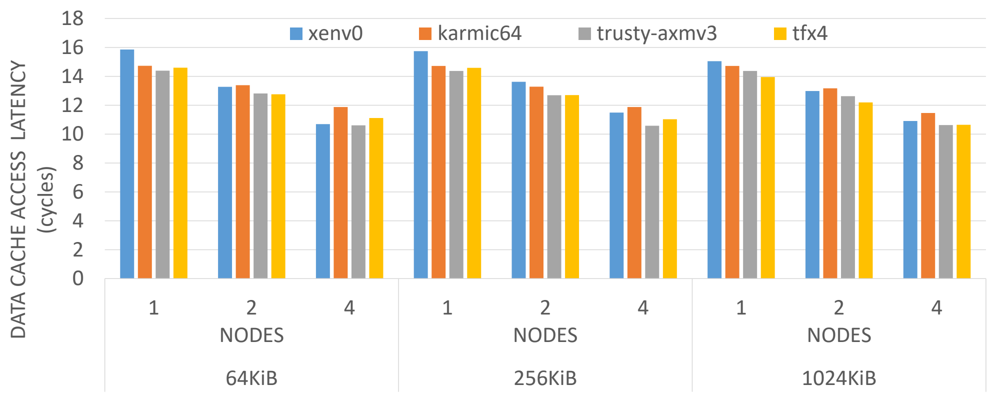


Figure 3.5: *Evaluation of the Data Cache Access Latency in the COTSon framework when using the Matrix Multiplication benchmark by varying cache size, number of nodes of the distributed system and different Linux distribution (xenv0=Ubuntu 16.04, karmic64=Ubuntu9.10, trusty-axmv3=Ubuntu 14.04, tfx4=Ubuntu 10.10). The DSE shows that the data-cache access latency is almost similar in each Linux distribution, but it is lowering when we increase the number of nodes.*

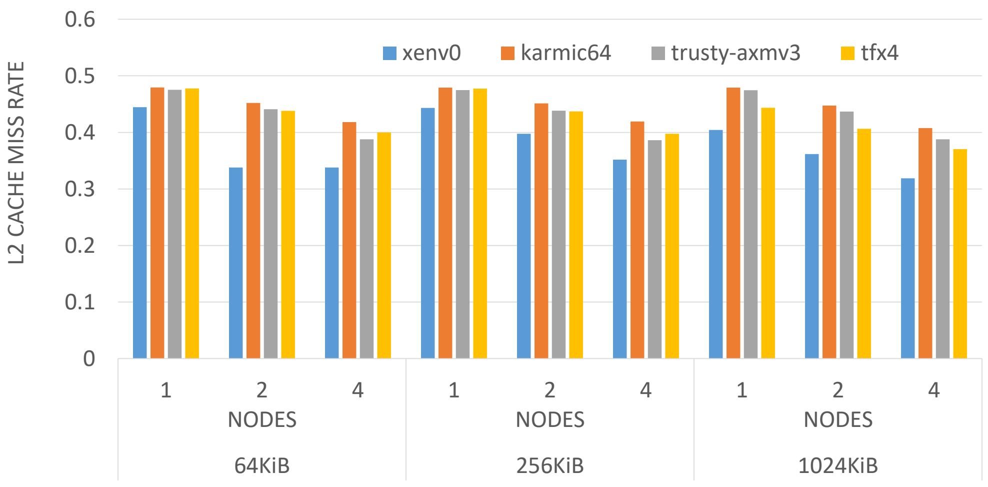


Figure 3.6: *Evaluation of the L2 Cache miss rate in the COTSon framework when using the Matrix Multiplication benchmark by varying cache sizes, number of nodes of the distributed system and different Linux distribution (xenv0=Ubuntu 16.04, karmic64=Ubuntu9.10, trusty-axmv3=Ubuntu 14.04, tfx4=Ubuntu 10.10). The Linux distribution xenv0 shows the lowest L2 miss rate compared to the other Linux distributions for all the presented configurations.*

However, the important information for us was to confirm the scaling of the DF-Threads model, while we increase the number of nodes/boards. It is possible to observe that the number of cycles is decreasing almost linearly - except the case of xenv0, which is decreasing sub-linearly - when we use two and four nodes compared to the case of a single board/node (Figure 3.9). Moreover, it is possible to understand the size of the cache that should be used in the physical system in order to properly accommodate the working set of our applications.

We further explored the reasons why we can obtain good scaling in the execution time with more nodes by analyzing the behavior of the data cache access latency (see Figure 3.5) and the L2 cache miss rate (see Figure 3.6). The data cache access latency allows us to evaluate the overall utilization of the entire cache hierarchy by combining the number of accesses/misses with the latencies specified into the COTSon simulator (see Table 3.2).

Again, this type of measurement was conveniently done in the simulator, while it is more difficult to perform it by directly using a High-Level Syn-

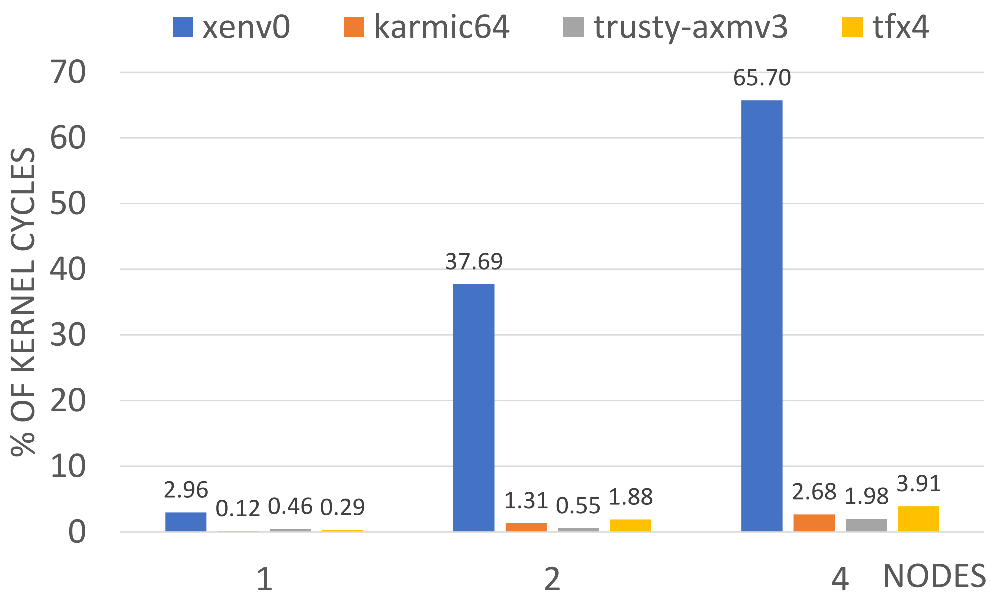


Figure 3.7: *Percentage of kernel cycles (over total number of cycles) in the COTSon framework when using the Matrix Multiplication benchmark with 512 as matrix size and 32KiB as cache size. We varied the number of nodes of the distributed system and the Linux distribution (xenv0=Ubuntu 16.04, karmic64=Ubuntu9.10, trusty-axmv3=Ubuntu 14.04, tfx4=Ubuntu 10.10). This DSE test permitted to detect a much larger kernel activity of the xenv0 distribution compared to the other three Linux distributions both in a single-node or multiple-node configurations.*

thesis framework (e.g., Xilinx HLS), especially when it comes to design a soft-processor and choose the best configuration (e.g., size of L2-cache and OS). In particular, we varied the number of nodes (1, 2, and 4), the OS distribution, and the cache size for L2 (64KiB, 256KiB, 1024KiB, to allow a wide exploration of the L2 cache). As we can see from Figure 3.6, the L2-cache miss rate is decreasing for all OS distributions while we vary the number of nodes, thus confirming that this is one of the main factors of the improvement of the execution time. The DSE shows that the data cache access latency is almost similar in each Linux distribution, but it is lowering when we increase the number of nodes. Thus, multiple nodes configuration has better utilization of the cache hierarchy, and it can be more convenient in the DF-Threads execution model. The decreasing of the data cache access latency when we

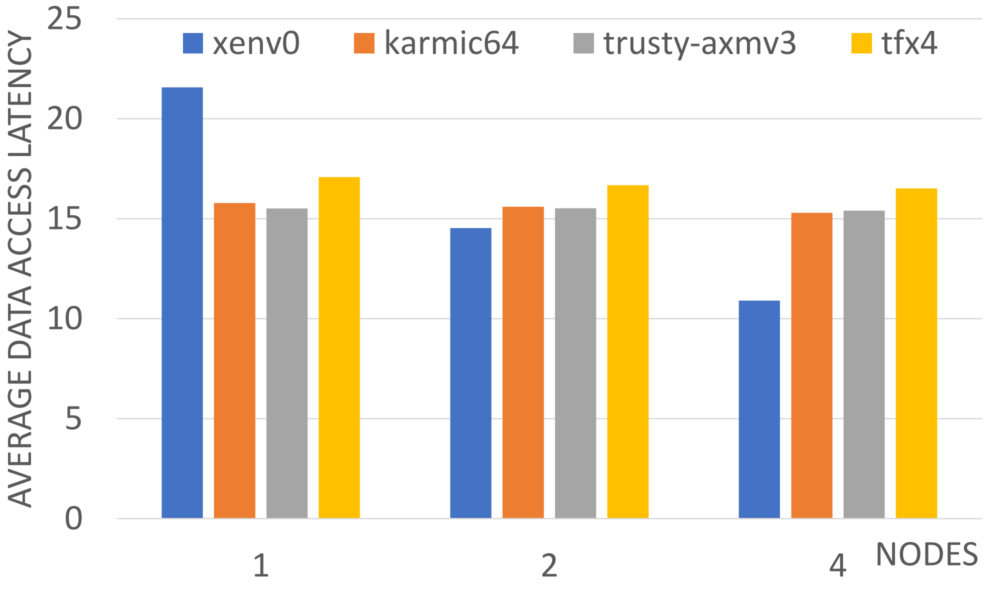


Figure 3.8: *Average data latency in the COTSon framework when using the Matrix Multiplication benchmark with 512 as matrix size and 32k as cache size. We varied the number of nodes of the distributed system and the Linux distribution (xenv0=Ubuntu 16.04, karmic64=Ubuntu9.10, trusty-axmv3=Ubuntu 14.04, tfx4=Ubuntu 10.10). The data access latency of xenv0is improved when we have more nodes. This improvement has less impact on total cycles (Figure 3.9) than the impact of kernel activity (Figure 3.7).*

increase the number of nodes also demonstrates the ability of the DF-Threads execution model in exploiting the data locality (Figure 3.5). Moreover, we can analyze which OS distribution leads to the best performance. For example, the xenv0 produces a huge amount of kernel activity during the computation (Figure 3.7). However, the combined effect of the kernel activity (Figure 3.7) and the average data latency (Figure 3.8) - considering L1, L2, and L3 caches - may affect the total execution time (Figure 3.9) quite heavily. Thanks to this preliminary DSE, we found that the OS distribution with the best tradeoff between memory accesses and kernel utilization is the trusty-axmv3.

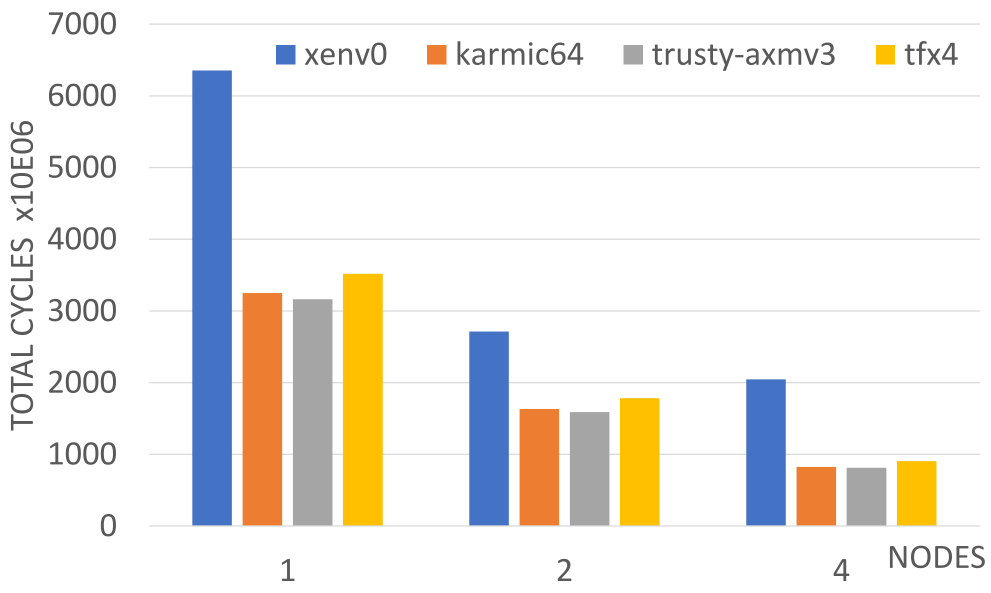


Figure 3.9: *Total number of cycles in the COTSon framework when using the Matrix Multiplication benchmark with 512 as matrix size and 32k as cache size. We varied the number of nodes of the distributed system and the Linux distribution (xenv0=Ubuntu 16.04, karmic64=Ubuntu9.10, trusty-axmv3=Ubuntu 14.04, tfx4=Ubuntu 10.10). The DSE allows us to detect that the four Linux distributions permit us to obtain good scalability when we increase the number of nodes. However, the xenv0 confirms the worst performance in terms of executed cycles, due to the huge number of kernel cycles shown in Figure 3.7.*

3.6 Final Remarks

The Data-Flow execution model is a viable paradigm to be explored today to achieve a high degree of parallelism in the modern multi-core/node architectures. In this chapter, we presented how the DF-Threads execution model can bridge the Data-Flow execution to a simple C-based programming model through the DF-Threads API.

Furthermore, as related research has discovered, Operating System (OS) increasingly affects the performance when the number of nodes increases. In order to study this impact, a simulator that can support both OS and multi-node simulation is valuable. We found that the OS impact could vary among different types of Linux distributions. We compare several Linux Ubuntu distributions to understand the performance sensitivity when accounting for all the software components like run-time and OS.

As one benchmark of high interest nowadays is Matrix Multiplication (a fundamental operation, e.g., the Deep Neural Networks), and we chose it as a simple driver for our experiments in order to study the OS impact on performance while varying key metrics of the architecture (e.g., L2 Cache size) and increasing the number of nodes of the distributed environment. We extract and analyze the key metrics like L2 Cache Miss Rate, execution cycles, data cache access latency, and kernel cycles for different Linux distributions. We showed the behavior of OS varying the L2 Cache size, reaching up to 60% in performance variations due to the different types of configurations and daemons installed.

Chapter 4

FPGA-Based Data-Flow Execution Engine for Heterogeneous Architectures

Recently, there has been a huge effort by the engineering communities to improve the parallelism of programming models with thread management such as P-threads, Cilk++, OpenMP, OpenMPI [13]. But in most of these models, synchronization, and distribution of thread's data between cores of different nodes need to be managed manually by programmers and imposes an extra effort [111, 13]. Instead, the Data-Flow execution model proposes better scalability by re-managing the distribution of many threads based upon the Data-Flow paradigm [3, 112]. The management of many threads across multiple core/node must consider the order of execution and the quantum time per thread, and the associations of allocation of a thread on a given core. This aspect begins to be serious as the entire system grows in complexity, memory hierarchies, interconnects, and distributed resources. In this chapter, we describe and evaluate the possibility of reducing such inefficient with the adoption of a Data-Flow execution model assisted by a DF-Engine, implemented at the hardware level, for accelerating the function execution and the distribution of many threads in a distributed environment composed of heterogeneous boards (e.g., AXIOM-Board).

4.1 DF-Threads Management

The Data-Flow execution paradigm can be exploited either completely on hardware or it can be used in a control flow processor to improve the execution time by Thread Level Parallelism (TLP) [33, 73]. The main task of a Data-Flow scheduler is to materialize TLP in such a way that respects the Data-Flow paradigm [113, 114]. A hardware DF-Engine, which can manage an appropriate distribution of the thread's data across the cores of a distributed system, may allow keeping a shared-memory programming model while distributing the computation.

The two main actors of the proposed execution model are i) the Processing System (PS), the control flow processor, and ii) the DF-Threads Hardware Scheduler (DTHS) implemented into the Programmable Logic (PL) (e.g., the FPGA). The PS is responsible for creating and executing the Data-Flow threads. Whenever a new DF-Thread is needed, the HS is responsible for setting the meta-information of that thread and stores them into an associated memory segment called "frame". When a producer DF-Thread wants to write its outputs into the consumer DF-Thread, the DTHS performs the writing in a lazy and asynchronous way, without blocking the PS. After the producer has written their outputs, the DTHS decreases the synchronization count (SC) of the consumer DF-Thread. If the SC is equal to zero, the DF-Thread is ready to execute and it is moved into a FIFO queue named Global Ready Queue(GRQ). When the PS asks for a new DF-Thread to execute, the DTHS dequeue the first element of the RQ and sends it to the PS.

The scheduling approach adopted in the implementation tested into the COTSon simulator is based on a hierarchical task queuing. As depicted in the Figure 4.1, each core of the distributed system has a small Local Ready Queue (LRQ) on which it primarily operates. The LRQ is accessible only by the core, avoiding the usage of lock. While the GRQ is placed in the global shared memory and managed only by the DTHS. When an LRQ of a core is quite empty, the DTHS is responsible for moving DF-Threads from the GRQ to the LRQ [54, 115, 116].

Figure 4.2 shows the block designs exploited to materialize and map the DF-Threads management on the PL part of the AXIOM-Board as an individual soft IP. The DTHS is tightly coupled (e.g., based on the AXI Stream protocol and proper buffering) to the NIC [117] module to be able to transceive appropriate messages to distribute the workloads across the network.

Since the DTHS is offloaded on the PL, all overheads regarding the thread management are reduced. As such, this leads to a better distribution of tasks through the PL as well.

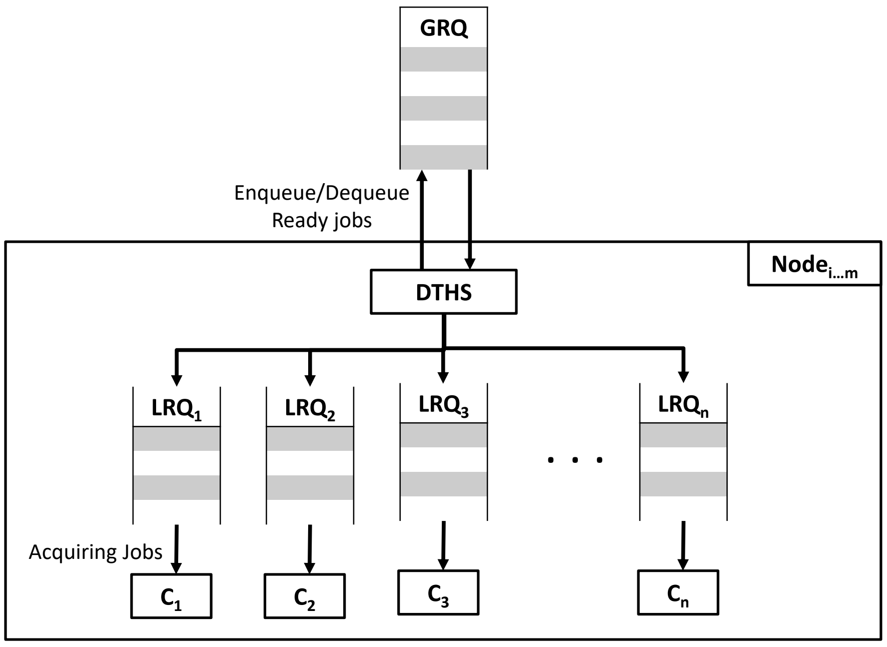


Figure 4.1: *Scheduling technique implemented for the proposed DF-Threads execution model, where the DF-Treads Hardware Threads Scheduler (DTHS) has the responsibility to enqueue/dequeue ready jobs Into the shared Global Ready Queue (GRQ). Each core (C) of the node has a small Local Private Queue (LRQ), which is filled by the DTHS when it is quite empty.*

4.2 From COTSon timing model to FPGA

The motivation of this section is to explain the workflow adopted to migrate the design of the DF-Threads execution model from the COTSon simulator to a heterogeneous architecture by using a simple and well-known driving example, i.e., the design of a two-way set-associative cache. First, the execution model has been designed in the COTSon simulator, then its correct functioning and the desired design goals have been tested. Finally, the timing description of the desired architecture has been transformed into a hardware description language in collaboration with my research group.

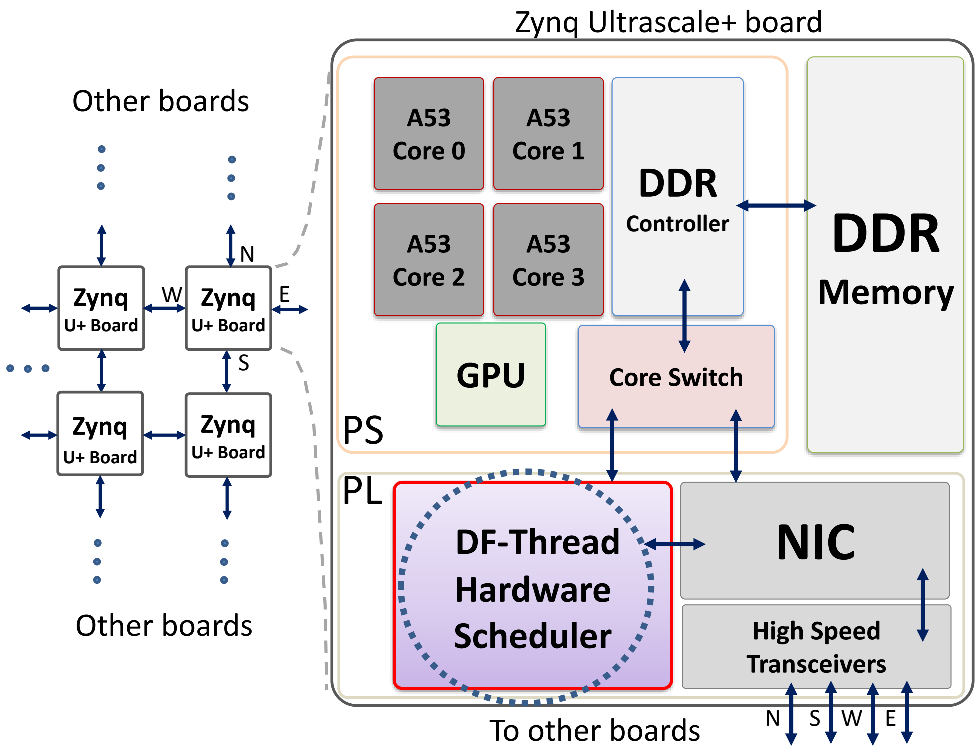
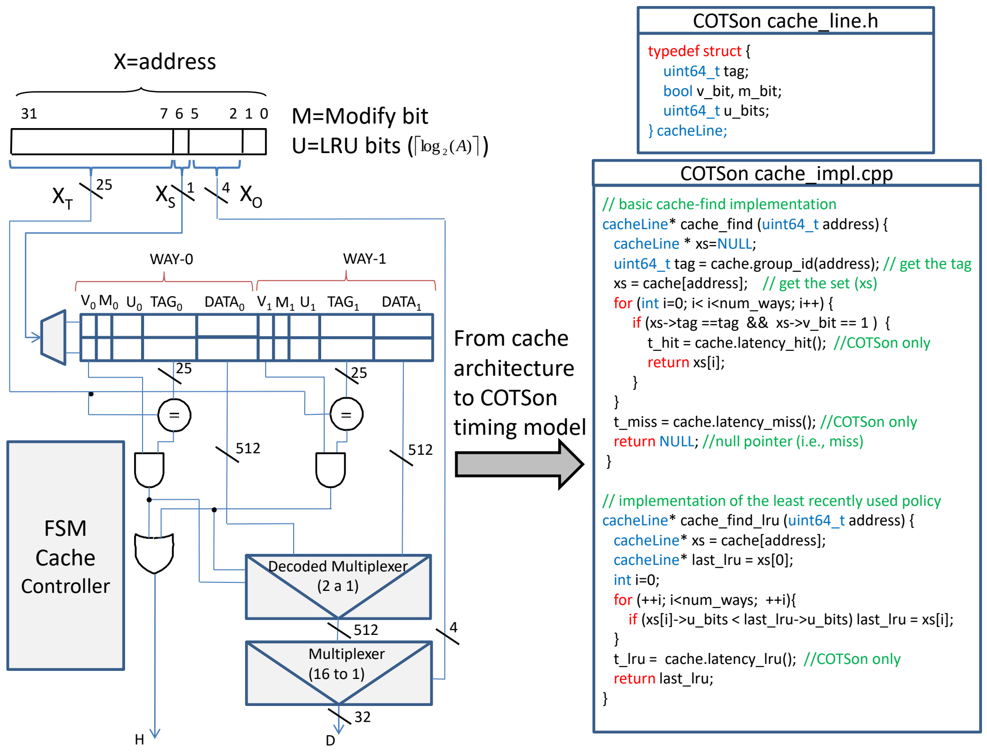


Figure 4.2: *Proposed scalable DF-Threads architecture mapped on the Zynq Ultrascale+ based board like AXIOM board (left side). The detailed bock designs for each node (board) are depicted on the right side. The proposed DF-Threads hardware scheduler is completely designed on PL (dotted circle). U+: Ultrascale+, PS: Processing System, PL: Programmable Logic, NIC: Network Interface Card [117].*

4.2.1 Timing Model Definition

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 [5]. 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 (see Figure 4.4) easily.


X=address
31
7 6 5
2 1 0
M=Modify bit
U=LRU bits ($\lceil \log_2(A) \rceil$)
X_T
25
X_S
1
4
X_O
WAY-0
WAY-1
V_0 M_0 U_0 TAG_0 $DATA_0$ V_1 M_1 U_1 TAG_1 $DATA_1$
25
25
=
=
512
512
FSM Cache Controller
Decoded Multiplexer (2 a 1)
512
4
Multiplexer (16 to 1)
32
H
D
From cache architecture to COTSon timing model
COTSon cache_line.h
COTSon cache_impl.cpp

```
typedef struct {
   uint64_t tag;
   bool v_bit, m_bit;
   uint64_t u_bits;
} cacheLine;
```

```
// basic cache-find implementation
cacheLine* cache_find (uint64_t address) {
  cacheLine * xs=NULL;
  uint64_t tag = cache.group_id(address); // get the tag
  xs = cache[address];    // get the set (xs)
  for (int i=0; i< i<num_ways; i++) {
     if (xs->tag ==tag  &&  xs->v_bit == 1 )  {
        t_hit = cache.latency_hit();  //COTSon only
        return xs[i];
     }
  }
  t_miss = cache.latency_miss(); //COTSon only
  return NULL; //null pointer (i.e., miss)
}

// implementation of the least recently used policy
cacheLine* cache_find_lru (uint64_t address) {
  cacheLine* xs = cache[address];
  cacheLine* last_lru = xs[0];
  int i=0;
  for (++i; i<num_ways;  ++i){
    if (xs[i]->u_bits < last_lru->u_bits) last_lru = xs[i];
  }
  t_lru =  cache.latency_lru();  //COTSon only
  return last_lru;
}
```

Figure 4.3: *Example of logic a 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) are 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.*

After defining the model, we can simulate and measure the performance of it.

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. 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.

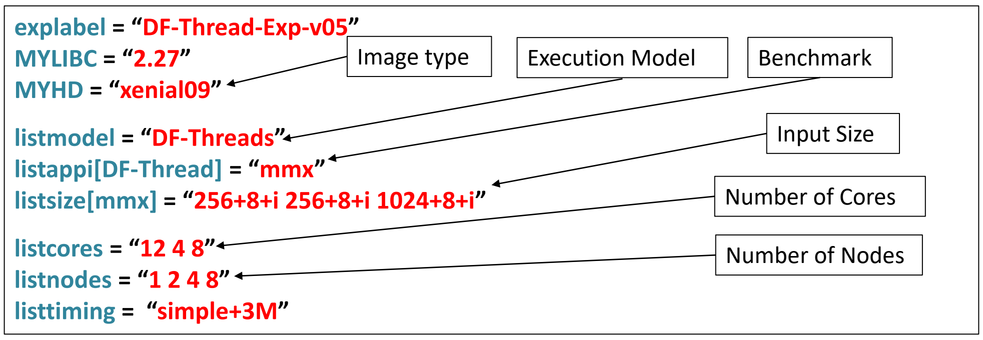
When we want to read or write data, which are stored in a byte address (X in Figure 4.3), we check if the data are already present in the cache. The cache controller implements the algorithm to find the data in the cache. Although not visible in the left part of Figure 4.3, there is a control part also 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, which is a simple linear search, and the other one named find_lru.

From the timing model of the implemented cache in COTSon, we migrate the design into a High-Level Synthesis framework (e.g., Xilinx HLS tools).

The advantage of using a hybrid methodology is that the DSE of the architecture of this small cache takes a few seconds in the COTSon, while it takes approximately four hours on a powerful workstation to synthesize and perform the DSE with a hardware design tool (e.g., Xilinx HLS tools).

4.2.2 COTSon configuration and Timing

One more feature of the DSE toolset is the capability of easily integrating the modeled components (e.g., the simple two-way set-associative cache of the previous subsection). As we can see in Figure 4.5, it is possible to build the overall architecture by specifying how to integrate the component in a higher-level configuration file ("Level-2" in Figure 4.6, the "MYDSE" configuration file). In particular, we define the following simple syntax: the character "-" is the link between two architectural COTSon blocks and the "+" character separates different links between such blocks. The architectural blocks



```
explabel = "DF-Thread-Exp-v05"
MYLIBC = "2.27"
MYHD = "xenial09"

listmodel = "DF-Threads"
listappi[DF-Thread] = "mmx"
listsize[mmx] = "256+8+i 256+8+i 1024+8+i"

listcores = "12 4 8"
listnodes = "1 2 4 8"
listtiming =  "simple+3M"
```

Figure 4.4: *INFOFILE example, which describes a Design Space Exploration experiment.*

are implicitly defined since they appear in the link specification. The "." character serves to replicate a set of architectural blocks, which follow the ".", for "m" times, where "m" is the number of cores. This is shown in the "listarch" variable of Figure 4.5: i.e., the part "l2-ic+l2-dc+ic-cpu+dc-cpu" will be instantiated "m" times.

As depicted in Figure 4.6, at the higher level, it is possible to specify the parameters in an even compact way, and there is the ability to indicate several instances of such parameters so that MYDSE can generate the design space points to be explored. In the COTSon configuration, the MYDSE points will be assigned to the parameter of the corresponding architectural element. Moreover, it is possible to specify the latencies of an architectural block, which are used by its timing model for the execution time estimation. In the next section, we present how, thanks to the described methodology, we were able to reduce significantly the DSE cycles and development time of a relatively large project like the AXIOM project, and produce reliable specifications to be implemented on the AXIOM board.

4.3 Generalization to the AXIOM project

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 [13, 11]. The AXIOM project has defined a simple but powerful architecture

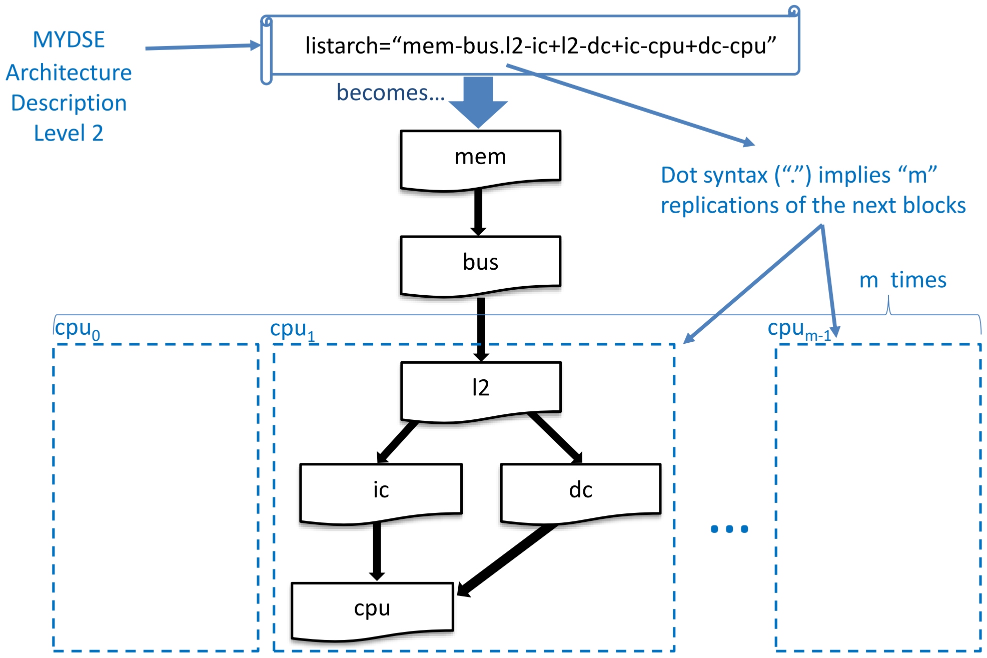
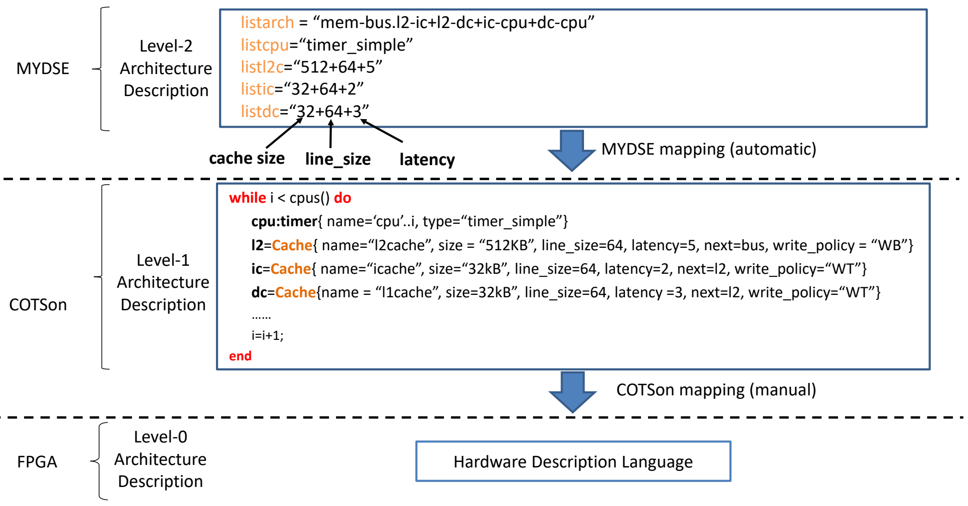

MYDSE Architecture Description Level 2
listarch="mem-bus.l2-ic+l2-dc+ic-cpu+dc-cpu"
becomes...
mem
bus
Dot syntax (".") implies "m" replications of the next blocks
m times
cpu_0
cpu_1
cpu_{m-1}
l2
ic
dc
cpu
...

Figure 4.5: *Level 2 architecture description of the cache model in COTSon by using MYDSE toolset. In this design, the CPU is directly connected to both an Instruction Cache (ic) and a Data Cache (dc). The ic and dc caches are then connected to another level of caching, the L2-Cache (l2), which is connected to the main memory (mem) through the bus.*

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 the programming model.

A Single Board Computer (SBC), named "AXIOM-Board", has been developed at the beginning of 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 a wide range of scenarios like Artificial Intelligence, Smart Home Living, Smart Video Surveillance, just to name a few.


MYDSE
Level-2 Architecture Description
cache size
line_size
latency
MYDSE mapping (automatic)
COTSon
Level-1 Architecture Description
COTSon mapping (manual)
FPGA
Level-0 Architecture Description
Hardware Description Language

```
listarch = “mem-bus.l2-ic+l2-dc+ic-cpu+dc-cpu”
listcpu=“timer_simple”
listl2c=“512+64+5”
listic=“32+64+2”
listdc=“32+64+3”
```

```
while i < cpus() do
    cpu:timer{ name=‘cpu’..i, type=“timer_simple”}
    l2=Cache{ name=“l2cache”, size = “512KB”, line_size=64, latency=5, next=bus, write_policy = “WB”}
    ic=Cache{ name=“icache”, size=“32kB”, line_size=64, latency=2, next=l2, write_policy=“WT”}
    dc=Cache{name = “l1cache”, size=32kB”, line_size=64, latency =3, next=l2, write_policy=“WT”}
    ……
    i=i+1;
end
```

Figure 4.6: *The relation between the higher level MYDSE description, the COTSon configuration file and the final hardware translation: at the higher level, we specify the parameters in a compact way (level-2 architecture description), and we can indicate several instances of such parameters so that MYDSE can generate the design space points to be explored. In the COTSon configuration (level-1), the MYDSE points will be automatically mapped to the parameter of the corresponding architectural element (bottom part of the figure: the next field specifies the position in the architecture tree, WB means write-back and, WT is the write-through policy). Finally, the architecture description is mapped manually from the COTSon description to a Hardware Description Language (i.e., Xilinx HLS tools).*

4.3.1 The AXIOM Board

The main goal 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 industrial project partners in order to fulfill the needs of future Cyber-Physical Systems (CPS). In this respect, partners SECO, EVIDENCE, FORTH, BSC, VIMAR, HERTA defined use case scenarios and compared the AXIOM concept against possible exploitation paths and use case scenarios that have been carried out to assess the features of the AXIOM board. As such, the inferred information has been translated into the definition of the AXIOM architecture, its external interface, the FPGA type, and its functional requirements.

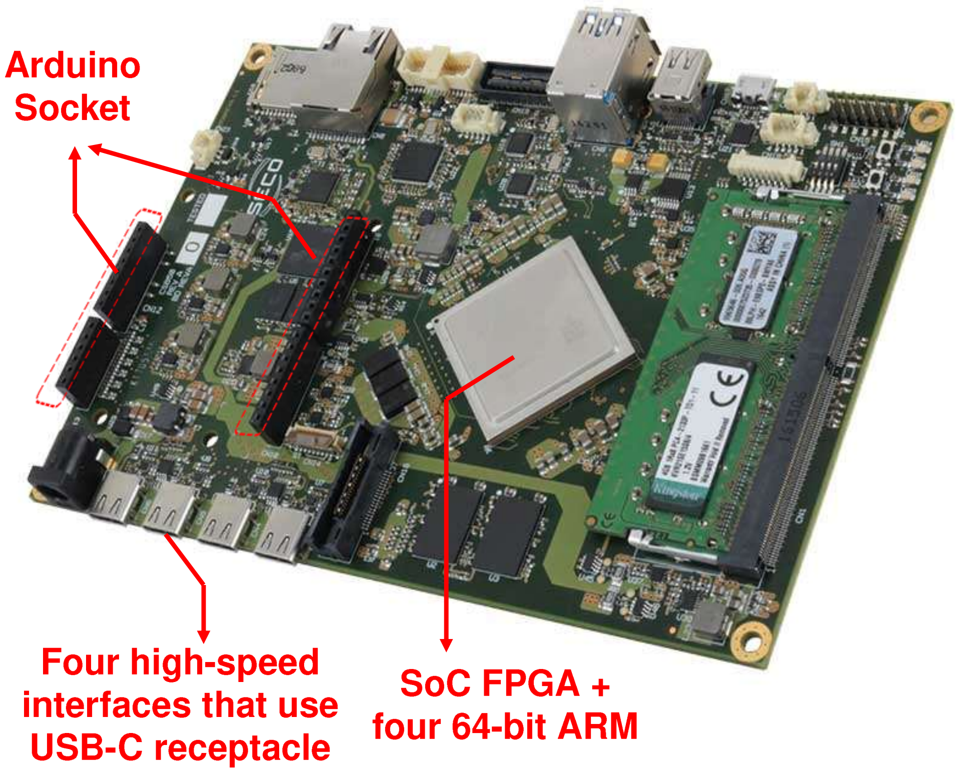

Arduino Socket
Four high-speed interfaces that use USB-C receptacle
SoC FPGA + four 64-bit ARM

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.*

For instance, modular scalability is enabled by high throughput and low latency interconnect, which made the partners choose one of the advanced FPGA platforms available at the proceeding time of the project, which is the Xilinx MPSoC Zynq Ultrascale+ (ZU9EG) platform. Consequently, this opens the possibility to reach a design that is capable of being efficiently interfaced with the physical world and be used in smart applications with much more reliability and energy efficiency [118]. The FPGA, in that sense, provides the appropriate substrate for the integration of our key features, providing the customized and reconfigurable acceleration as well as the implications to have a high-speed board-to-board communication.

We briefly describe here the main specification of the AXIOM board manufactured by SECO company, the main pillars of the architecture (Hardware-Software) is based as follows:

1. MPSoC 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
3. Lower-Level Thread Scheduler
4. High-speed, inexpensive interconnects managed by an efficient Network Interface Card (NIC) [117] module to allow scalability and realization of the parallel programming models as well
5. Efficient interfaces for the Cyber-Physical world, such as Arduino connectors (to be able to interface with sensors and actuators), USB, Ethernet.

Hardware Specification

The adopted platform is the Xilinx ZU9EG platform, including a 64-bit quad ARM Cortex-A53 based GPPs with working frequency up to 1.5GHz, with Single/Double Precision Floating Point; 32KB L1 Cache and 1MB L2 Cache. This can support a number of activities such as the OS (or system tasks) but also whenever there is a sequential task that invokes Instruction Level Parallelism (ILP) rather than other forms of acceleration. 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, 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 16.3 Gb/s throughput. These transceivers are exposed to the physical world through the USB-C receptacle (with AXIOM protocol), which is more easy to be used due to its two-fold rotationally-symmetrical connector.

Open-source Software stack

Moreover, the recent success of Single Board Computers (SBCs) such as the UDOO [119], and RaspberryPi further addressed the necessities for using open-source software. However, there is not yet a concrete agreement on which parallel programming model is the best.

The AXIOM software architecture [36, 120] includes the techniques used for managing the memory in the cluster, the node interconnections, and the software stack used to manage the cluster, which is in turn composed of the device driver and the libraries for board-to-board communication and memory allocation.

Network Interface Card (NIC)

In order to provide acceleration functions in general aspects of the applications, the uses of OS and the programming model, a NIC has been designed and implemented on the PL [117]. This leads to efficient and scalable program execution and seamless interconnection of systems spanning multiple boards. Such connectivity permits designers to expand and scale-up their system by interconnecting more boards in a flexible and low-cost manner, without the need for expensive particular cables and connectors.

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

4.3.2 From COTSon Distributed System to AXIOM Distributed System

The aim of the AXIOM project was to define a software/hardware architecture configuration, to build scalable embedded systems, which could allow a distributed computation across several boards by using a transparent scalable method like the DF-Threads [3, 121, 76].

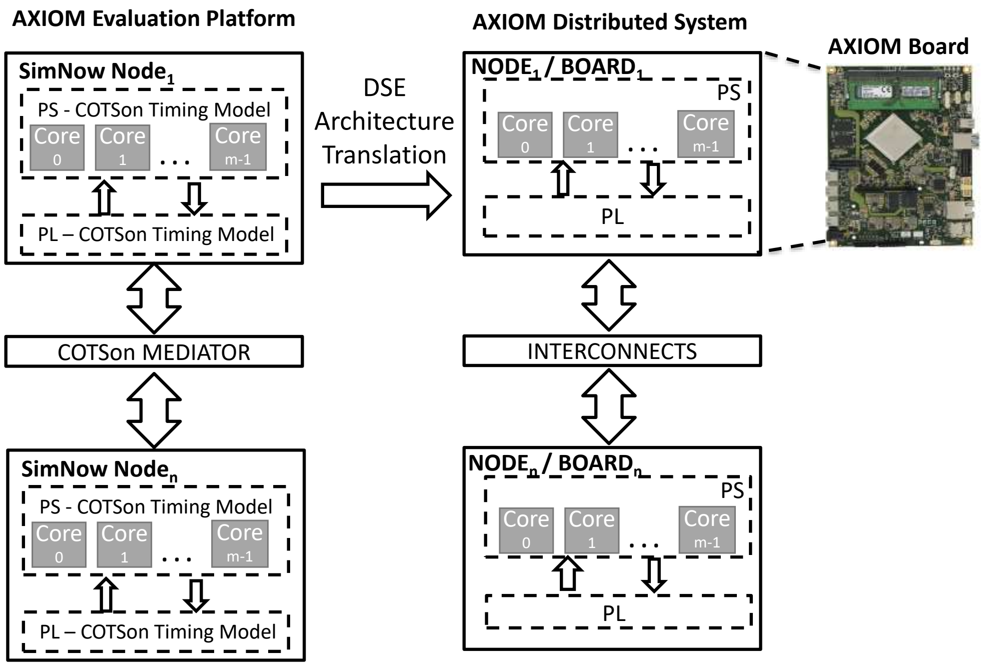


Figure 4.8: *From COTSon Distributed System definition to AXIOM Distributed System by using the DSE Tools. The Processing System (PS), the Programmable Logic (PL) and the Interconnects of the AXIOM Board are simulated and evaluated into the COTSon framework with the definition of the respective timing models.*

In order to achieve this goal, we rely on RDMA capabilities and a full operating system (OS) to interact with the OS scheduler, memory management and other system resources. Following the methodology illustrated in this thesis, it is possible to include the effects of all these features thanks to the

COTSon+MYDSE full-system simulation framework. After the desired software and hardware architecture were selected in the simulation framework, we started the migration to the physical hardware. we had clear that we needed at least the following features:

i) Possibility to exchange data frames rapidly via RDMA across several boards: this could be implemented in hardware thanks to the FPGA high-speed transceivers;

ii) 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.

Basing on these requirements, it was possible to first prototype the software implementation and the hardware design into a timing model of the COTSon simulator. Then, the MYDSE toolset has been used to find the best configuration in terms of performance (e.g., execution time, load balancing). Finally, we migrated the final implementation into the AXIOM Distributed Environment (Figure 4.8 - right).

4.4 Experimental Results

For the sake of this initial exploration, two simple benchmarks have been chosen for the evaluation of the DF-Threads: Recursive Fibonacci and Matrix Multiplication.

- **Recursive Fibonacci** (RFIB) benchmark has been chosen to stress the thread management and quickly evaluate the performance, while there is a need for scheduling many threads. This benchmark takes as input the n of Fibonacci and a threshold that stops the generation of the parallel recursive calls.

- **Blocked Matrix Multiplication** (MM) benchmark involves more memory operations than RFIB and also is a widely used kernel in machine learning. The Matrix Multiplication implementation is the blocked version, where a matrix is partitioned in multiple sub-matrices, or blocks, according to the block size that is set. The measurements focus only on the computational Region of Interest (ROI) of the benchmark as it is the usual practice in comparisons. The Matrix Multiply algorithm used to evaluate OpenMPI and CUDA is the standard available version for such programming models. The results of the benchmarks are checked for correctness at the end of the run. Multiple runs (at least 5) have also been repeated to reduce possible statistical oscillations.

4.4.1 Recursive Fibonacci

The evaluation of the Recursive Fibonacci benchmark has an input size of 35 and 13 as a threshold, by varying the core number of the distributed system up to 16. Different configurations of Fibonacci number and threshold have been tested without noticed variation in the different experiments, and the input size is chosen in respect of the simulation time.

As can be seen in Figure 4.9, DF-Threads show a good degree of scalability. The results confirm that the scheduler of the DF-Threads can handle and distribute properly many fine-grain threads among multiple nodes. CUDA and OpenMPI have not been evaluated with the Fibonacci benchmark due to their poor effectiveness with recursive execution on such platforms.

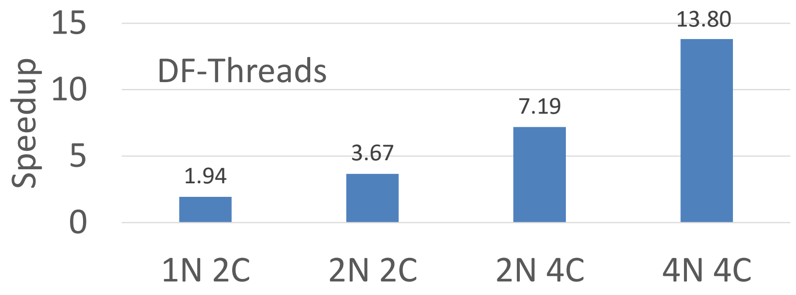


Figure 4.9: *Speedup of the Recursive Fibonacci benchmark with input size 35 and threshold 13, by using the DF-Threads and by varying the number of nodes (N) and cores (C) of the distributed system.*

4.4.2 Block Matrix Multiply

The GFLOPS/core metric is used to compare the performance of the DF-Threads with OpenMPI and CUDA, due to the current lower number of cores of our Distributed System with respect to CUDA. The matrix type is squared with 448 as size (to avoid a multiple of a power of two, which may cause multiple cache conflicts on a few cache lines) and with 8 as the block size. Different types of block sizes have been tested (4,8,16,32) without showing variation in results, and the best trade-off between matrix size and threads generated is chosen.

As we can see in Figure 4.10, the GFLOPS/core of the DF-Threads outperforms both CUDA and OpenMPI.

OpenMPI is outperformed by a large factor of about 5.5x in the case of 1 core (1N) or about 140x in the case of 16 core (16C). This is due to several factors: first of all, the OpenMPI runtime library represents a wide middleware layer; secondly, there is the need of invoking system calls that in turn may need a time consuming operating system activity to move content buffer and manage the send and receive operations on the physical media. In the DF-Threads such overheads are reduced, thanks to the simpler interface and the hardware management of the data frames and thread metadata. As depicted in Figure 4.11, the kernel activity of the DF-Threads is quite limited in comparison with OpenMPI. The OpenMPI version used in this test is the 1.10.2, which is contained in the official repositories of Ubuntu 16.04 LTS.

CUDA is outperformed by almost a factor of 1.7x among all configurations of the distributed system, due to the efficiency of our scheduling mechanisms. The CUDA board is the Tesla-C1060 with 240 CUDA core, 610MHz GPU clock and 4GiB of RAM. The CUDA platform used is one of the first Tesla boards available on the market, but the proposed implementation of DF-Threads is also at the first version and it is not yet optimized. Therefore, the DF-Threads is capable of exploiting better the resources of the distributed system, also thanks to the parallelism exposed by the Data-Flow mechanism and the DF-Thread API.

Moreover, we think that there is still much space for further optimization of the DF-Threads. For example, the data locality could be improved, and,

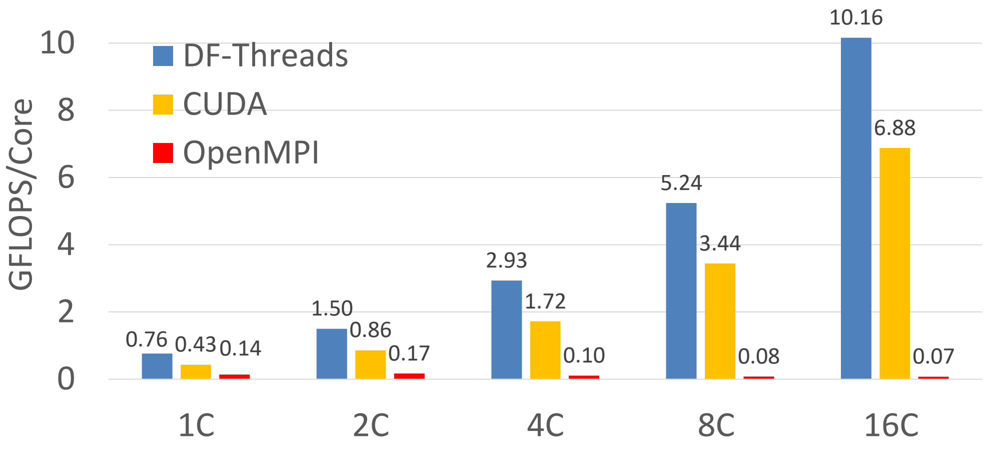


Figure 4.10: *GFLOPS/core comparison between DF-Threads, OpenMPI and CUDA by using the block Matrix Multiply benchmark with 448x448 as matrix size and 8 as block size. The number of nodes (N) varies from 1 to 16.*

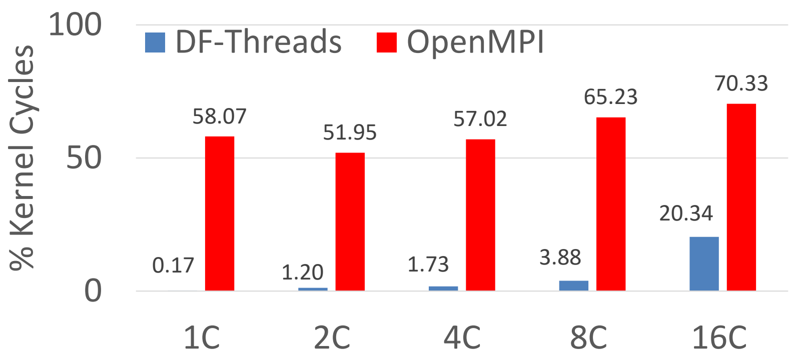


Figure 4.11: *DF-Threads and OpenMPI kernel cycles (left) for the block Matrix Multiplication benchmark with matrix size 448x448 and block size 8. The number of nodes (N) varies from 1 to 16.*

in collaboration with my research group, we are investigating a pre-fetching policy, which could load the data into the cache before starting the execution of the DF-Threads.

4.5 Final Remarks

In this Chapter, the workflow adopted by our research group is presented for developing an architecture that could be controlled by the designer in order to match the desired key performance metrics.

The main features of the COTSon simulator and the MYDSE toolset have been illustrated. Thanks to the functional-directed approach of the COTSon simulator, it is possible to define the behavior of any architectural components (e.g., a cache, TLB, Branch Predictor) for an early DSE and migrate in hardware only the selected architecture. The DSE toolset facilitates the modeling of architectural components in the earlier stages of the design.

We described the simple example of defining a two-way set-associative cache through the timing model of COTSon. After, we illustrated that the timing description made in the COTSon simulator is conveniently close to a High-Level Synthesis framework (e.g., Xilinx HLS), which reduces the required time for the DSE phase from hours to few seconds.

By using the workflow presented in this paper, our research group was able to successfully prototype the preliminary design of the DF-Threads for a heterogeneous architecture 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/).

We compared the DF-Threads with OpenMPI and CUDA by using the block Matrix Multiplication benchmark, where DF-Threads outperform both OpenMPI and CUDA in terms of GFLOPS/core.

Chapter 5

Conclusions and Future Works

The work presented in this thesis aims at finding an alternative solution for overcoming scalability problems in modern architectures. The full parallelism of such systems has not yet been completely exploited due to both execution model and programming model limitations. The Data-Flow execution model can overcome these limitations with the capability of creating many asynchronous fine-grain threads which they can run in parallel on a multi-core/node system [16, 31, 32, 44].

5.1 Summary

Further implementation and validation of the DF-Threads for a distributed embedded system are presented. The main idea is to offload a portion of an algorithm code to a DF-Engine (e.g., write output, thread distribution).

Directly design the implementation in hardware can be time-consuming. The Design Space Exploration plays an important role in evaluating different design options and reduces the development time from days/weeks to minute/hours. The Design Space Exploration toolset (e.g., COTSon + MYDSE) illustrated in this thesis permits our research group to design and evaluate complex architecture for high-performance computing (e.g., with 1000 general-purpose cores in the TERAFLUX project [55]), and model features of architectures which are not yet available on the market in a rapid way. Moreover, it is possible to use different types of programming models (e.g., OpenMPI, Cilk++, DSM) and execution models (e.g., von-Neumann, Data-Flow).

Since the proposed Data-Flow execution model is designed for the execution on the AXIOM-Board (AArch64 based), the COTSon simulator execution is validated against real architecture like x86_64 and AArch64 during the AXIOM project. The simulation framework has been extended with a tracing system for analyzing in detail the execution flow and check its correctness. The validation study confirms that the COTSon simulator is capable of simulating both x86_64 and AArch64 and obtaining results very close to the real hardware with four different benchmarks (Matrix Multiplication, Recursive Fibonacci, Cholesky Factorization, Radix Sort). Thanks to this validation, the COTSon simulator gives us the possibility to quickly explore the design space for the implementation of the proposed Data-Flow execution model.

The performance evaluation of the presented DF-Threads implementation required to extract and analyze possible overhead coming from external sources like Operating System. The COTSon simulator is a full-system simulator, which is able to run and observe the behavior of the Operating System activity. The results showed us the possible impact of four different Linux distributions on the execution, where the percentage of the kernel instruction can reach up to 60%. Moreover, we explored different configurations of the cache hierarchy and were able to observe the ability of our Data-Flow execution model to obtain a better utilization of the memory system when the number of nodes is increased.

The "functional-directed" approach of the COTSon simulator permits us to specify the implementation of the proposed Data-Flow execution model through the definition of a timing model. The timing model description is conveniently close to a High-Level Synthesis framework (e.g., Xilinx HLS), which facilitated the translation of the implementation from the simulation framework to the AXIOM-Board.

The evaluation of the proposed implementation focused on stressing the scheduling task of the DF-Engine in handling and distributing many fine-grain DF-Thread. For this reason, we relied on the Recursive Fibonacci benchmark. We also studied the performance of the model with a data-intensive application for evaluating aspects like memory operations and data moments. We select the Matrix Multiplication algorithm as data-intensive benchmark, which is the main computational kernel of widely used applications (e.g., Artificial Intelligence, Smart Home Living, Smart Video Surveillance). The experiments presented in this thesis show the capability of the DF-Threads execution model to distribute and manage many fine-grain threads among multiple heterogeneous boards. Moreover, we compared the DF-Threads implementation with well-known parallel programming models like OpenMPI and CUDA by using the Matrix Multiplication benchmark. DF-Threads achieved better performance per-core compared to OpenMPI and CUDA. In particular, OpenMPI has a large portion of OS-kernel activity, which is slowing down its performance. On the contrary, the Operating System's impact on the DF-Threads execution is negligible, thanks to the combination of the lightweight API and the hardware scheduler. If we consider the GFLOPS per core, DF-Threads is also competitive with respect to CUDA.

5.2 Future Direction

The analysis performed in this thesis on the proposed implementation of the DF-Thread execution model permitted us to conclude that there is room for further improvements. The results in the Section 2.2.3 have shown quite good utilization of the memory hierarchy, mostly when we increase the number of nodes. However, the miss rate and data access latency are still high. This requires the investigation of data locality and prefetching techniques to be integrated into the design.

The number of benchmarks used in the evaluation of this thesis is limited and should be integrated with other applications. We plan to increase the benchmark set with a less regular application in order to have a complete evaluation of the proposed implementation, especially regarding the performance of the load balancing algorithm adopted. For example, we are trying to adopt different scheduling techniques (e.g., task stealing). In addition, the DF-Engine implemented at the hardware level should be compared with a software implementation to evaluate the actual benefit of the hardware solution.

The proposed Data-Flow execution model aims at target heterogeneous embedded systems, which requires an evaluation of the energy efficiency performance. We plan to measure the power consumption of the execution model in both AXIOM-Board and COTSon simulator.

Bibliography

[1] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, San Jose CA, USA, 2011, pp. 365–376.

[2] G. Moore, "Moores law," *Electronics Magazine*, vol. 38, no. 8, p. 114, 1965.

[3] 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*, Paris, France, 2014, pp. 60–65.

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

[5] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "COTSon: infrastructure for full system simulation," *ACM SIGOPS Operting Systems Review*, vol. 43, no. 1, pp. 52–61, 2009.

[6] 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)*, Manchester, UK, 2019, pp. 1–6.

[7] 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)*, Florence, Italy, 2019, pp. 1–6.

[8] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *ACM SIGARCH Computer Architecture News*, vol. 3, no. 4, 1975, pp. 126–132.

[9] J. B. Dennis, "Data flow supercomputers," *Computer*, pp. 48–56, 1980.

[10] N. P. Jouppi, C. Young, N. Patil, and D. Patterson, "A Domain-Specific Architecture for Deep Neural Networks," *Communiction ACM*, vol. 61, no. 9, 2018.

[11] P. Kogge and J. Shalf, "Exascale computing trends: Adjusting to the" new normal"'for computer architecture," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 16–26, 2013.

[12] W. M. Johnston, J. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM computing surveys (CSUR)*, vol. 36, no. 1, pp. 1–34, 2004.

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

[14] S. Zeng, J. M. M. Diaz, and S. Raskar, "Toward A High-Performance Emulation Platform for Brain-Inspired Intelligent SystemsExploring Dataflow-Based Execution Model and Beyond," in *Proceedings of IEEE Computer Software and Applications Conference (COMPSAC)*, vol. 2, 2019, pp. 628–633.

[15] K. M. Kavi, B. P. Buckles, and U. N. Bhat, "A formal definition of data flow graph models," *IEEE Transaction on Computers*, pp. 940–948, 1986.

[16] 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)*, Madeira, Portugal, 2015, pp. 526–529.

[17] W. A. Najjar, E. A. Lee, and G. R. Gao, "Advances in the Dataflow Computational Model," *Parallel Computing*, vol. 25, 1999.

[18] L. A. Marzulo, T. A. Alves, F. M. França, and V. S. Costa, "Couillard: Parallel programming via coarse-grained data-flow compilation," *Parallel Computing*, vol. 40, no. 10, pp. 661–680, 2014.

[19] C. Meenderinck and B. Juurlink, "Nexus: Hardware support for task-based programming," in *Proceedings of IEEE Euromicro Conference on Digital System Design (DSD)*, Oulu, Finland, 2011, pp. 442–445.

[20] J. Castrillon, R. Leupers, and G. Ascheid, "Maps: Mapping concurrent dataflow applications to heterogeneous mpsocs," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 527–545, 2013.

[21] R. Townsend, M. A. Kim, and S. A. Edwards, "From functional programs to pipelined dataflow circuits," in *Proceedings of the ACM International Conference on Compiler Construction*, Austin TX, USA, 2017, pp. 76–86.

[22] M. Steuwer, T. Remmelg, and C. 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)*, Austin TX, USA, 2017, pp. 74–85.

[23] S. Ertel, J. Adam, and J. 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*, Vienna, Austria, 2018, pp. 41–50.

[24] L. Verdoscia and R. Giorgi, "A Data-Flow Soft-Core Processor for Accelerating Scientific Calculation on FPGAs," *Mathematical Problems in Engineering*, vol. 2016, no. 1, pp. 1–21, 2016.

[25] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. 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*, Eugene, USA, 2014, p. 6.

[26] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. 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*, San Jose California, USA, 2011, pp. 64–69.

[27] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati, "Targeting distributed systems in fastflow," in *European Conference on Parallel Processing*. Springer, 2012, pp. 47–56.

[28] G. Matheou and P. Evripidou, "Data-Driven Concurrency for High Performance Computing," *ACM Transaction on Architecture and Code Optimization (TACO)*, vol. 14, no. 4, pp. 53:1–53:26, 2017.

[29] J. R. Gurd, "The manchester dataflow machine," *Computer Physics Communications*, vol. 37, no. 1-3, pp. 49–62, 1985.

[30] D. E. Culler and G. M. Papadopoulos, "The explicit token store," *Journal of Parallel and Distributed Computing*, vol. 10, no. 4, pp. 289–308, 1990.

[31] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion, "Hybrid dataflow/von-neumann architectures," *IEEE Trans. on Parallel and Distrib. Systems*, vol. 25, no. 6, pp. 1489–1509, June 2014.

[32] G. Matheou and P. Evripidou, "Data-driven concurrency for high performance computing," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 4, p. 53, 2017.

[33] 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)*, Gramado, Brasil, 2007, pp. 263–270.

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

[35] 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.

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

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

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

[39] 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)*, Samos, Greece, 2015, pp. 262–269.

[40] A. Filgueras, M. Vidal, M. Mateu, D. Jiménez-González, C. Alvarez, X. Martorell, E. Ayguadé, D. Theodoropoulos, D. Pnevmatikatos, P. Gai *et al.*, "The AXIOM project: Iot on heterogeneous embedded platforms," *IEEE Design & Test*, 2019.

[41] R. Giorgi, F. Khalili, and M. Procaccini, "Translating Timing into an Architecture: The Synergy of COTSon and HLS (Domain ExpertiseDesigning a Computer Architecture via HLS)," *International Journal of Reconfigurable Computing*, 2019.

[42] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of Spring Joint Computer Conference*, Atlantic City, New Jersey, 1967, pp. 483–485.

[43] J. L. Gustafson, “Reevaluating amdahl’s law,” *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[44] R. Giorgi and P. Bennati, “Reducing leakage in power-saving capable caches for embedded systems by using a filter cache,” in *Proceedings of ACM MEmory performance: DEaling with Applications, systems and architecture (MEDEA)*, Brasov, Romania, 2007, pp. 105–112.

[45] M. Budiu, P. V. Artigas, and S. C. Goldstein, “Dataflow: A complement to superscalar,” in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Austin TX, USA, 2005, pp. 177–186.

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

[47] 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)*, Edmonton, Canada, 2014, pp. 30–37.

[48] 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. Kolia, F. Li, M. Lujn, 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)*, Santander, Spain, 2013, pp. 272–279.

[49] S. Zuckerman, J. Arteaga, J. Suetterlein, H. Wei, E. Garcia, G. Gao, A. Scionti, and R. Giorgi, “Evaluation of the codelet runtime system on a teradevice,” Siena, Italy, pp. 1–53, 2014, deliverable. [Online]. Available: `http://www.dii.unisi.it/ giorgi/papers/Zuckerman14a.pdf`

[50] M. Procaccini and R. Giorgi, “X86_64 vs Aarch64 Performance Validation with COTSon,” pp. 1–4, 2018, poster. [Online]. Available: `http://www.dii.unisi.it/ giorgi/papers/Procaccini19-acaces.pdf`

[51] R. Giorgi and M. Procaccini, "Bridging a Data-Flow Execution Model to a Lightweight Programming Model," Dublin, Ireland, 2019, in press. [Online]. Available: `https://www3.diism.unisi.it/ giorgi/papers/Giorgi19-hpcs.pdf`

[52] 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*, Pavia, Italy, 2019, pp. 422–429.

[53] M. Procaccini and R. Giorgi, "Simulation infrastructure for the next kilo x86-64 chips," in *HiPEAC ACACES-2017*, Fiuggi, Italy, 2017, pp. 91–94, poster. [Online]. Available: `http://www.dii.unisi.it/ giorgi/papers/Procaccini17.pdf`

[54] M. Procaccini, F. Khalili, and R. Giorgi, "An fpga-based scalable hardware scheduler for data-flow models," in *HiPEAC ACACES-2018*, Fiuggi, Italy, July 2018, pp. 1–4, poster.

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

[56] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic, "RAMP: Research accelerator for multiple processors," *IEEE micro*, vol. 27, no. 2, pp. 46–57, 2007.

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

[58] 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. Kolia, J. Landwehr, N. L. Minh, F. Li, M. Lujn, 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*, vol. 38, no. 8, Part B, pp. 976–990, 2014.

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

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

[61] D. Theodoropoulos, S. Mazumdar, E. Ayguade, N. Bettin, J. Bueno, S. Ermini, A. Filgueras, D. Jiménez-González, C. Á. Martínez, X. Martorell *et al.*, "The AXIOM platform for next-generation cyber physical systems," *Microprocessors and Microsystems*, vol. 52, pp. 540–555, 2017.

[62] 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).* Samos, Greece: Springer, 2009, pp. 78–87.

[63] J. Chen, M. Annavaram, and M. Dubois, "Slacksim: a platform for parallel simulations of cmps on cmps," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 2, pp. 20–29, 2009.

[64] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *Computer*, no. 2, pp. 59–67, 2002.

[65] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.

[66] H. Zeng, M. Yourst, K. Ghose, and D. Ponomarev, "MPTLsim: a simulator for x86 multicore processors," in *Proceedings of IEEE Design Automation Conference*, San Francisco CA, USA, 2009, pp. 226–231.

[67] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: Towards real-time hypervisor scheduling in Xen," in *Proceedings of ACM International Conference on Embedded Software*, Taipei, Taiwan, 2011, pp. 39–48.

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

[69] J. E. Miller, H. Kasture *et al.*, "Graphite: A distributed parallel simulator for multicores," in *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Bangalore, India, 2010, pp. 1–12.

[70] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of ACM SIGPLAN notices*, vol. 40, no. 6, London, UK, 2005, pp. 190–200.

[71] 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)*, Madrid, Spain, 2014, pp. 264–269.

[72] A. Portero, Z. Yu, and R. Giorgi, “T-Star (T*): An x86-64 ISA Extension to support thread execution on many cores,” in *HiPEAC ACACES-2011*, Fiuggi, Italy, July 2011, pp. 277–280, poster. [Online]. Available: `http://www.dii.unisi.it/ giorgi/papers/Portero11b.pdf`

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

[74] E. Blem, J. Menon, and K. Sankaralingam, “Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures,” in *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Shenzhen, China, 2013, pp. 1–12.

[75] M. E. Thomadakis, “The architecture of the Nehalem processor and Nehalem-EP SMP platforms,” *Resource*, vol. 3, no. 2, pp. 30–32, 2011.

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

[77] J. Cong and B. Yuan, “Energy-efficient scheduling on heterogeneous multi-core architectures,” in *Proceedings of the ACM International Symposium on Low Power Electronics and Design (ISLPED)*, Redondo Beach CA, USA, 2012, pp. 345–350.

[78] G. Agosta, W. Fornaciari, G. Massari, A. Pupykina, F. Reghenzani, and M. Zanella, “Managing heterogeneous resources in hpc systems,” in *Proceedings of the ACM Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM)*, Manchester, UK, 2018, pp. 7–12.

[79] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer, “Libwater: heterogeneous distributed computing made easy,” in *Proceedings of the ACM International Conference on supercomputing (ICS)*, Eugene Oregon, USA, 2013, pp. 161–172.

[80] M. F. O'Boyle, Z. Wang, and D. Grewe, "Portable mapping of data parallel programs to opencl for heterogeneous systems," in *Proceedings of the IEEE International Symposium on Code Generation and Optimization (CGO)*, Shenzhen, China, 2013, pp. 1–10.

[81] P. Bellasi, G. Massari, and W. Fornaciari, "Effective runtime resource management using linux control groups with the barbequertrm framework," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 2, p. 39, 2015.

[82] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: Determinacy, termination, queueing," *SIAM Journal on Applied Mathematics*, vol. 14, no. 6, pp. 1390–1411, 1966.

[83] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic dataflow processor," in *Proceedings of the Symposium on Computer Architecture (ISCA)*, Austin TX, USA, 1974, pp. 126–132.

[84] J. B. Dennis, "The varieties of data flow computers," in *Advanced computer architecture*, 1986, pp. 51–60.

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

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

[87] R. Vedder and D. Finn, "The Hughes data flow multiprocessor: Architecture for efficient signal and data processing," *ACM SIGARCH Computer Architecture News*, vol. 13, no. 3, pp. 324–332, 1985.

[88] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester prototype dataflow computer," *Communications of the ACM*, vol. 28, no. 1, pp. 34–52, 1985.

[89] N. Ito, M. Sato, E. Kuno, and K. Rokusawa, "The architecture and preliminary evaluation results of the experimental parallel inference machine PIM-D," *ACM SIGARCH Computer Architecture News*, vol. 14, no. 2, pp. 149–156, 1986.

[90] Y. N. Patt, W.-m. Hwu, and M. Shebanow, "HPS, a new microarchitecture: rationale and introduction," *ACM SIGMICRO Newsletter*, vol. 16, no. 4, pp. 103–108, 1985.

[91] J. Silc, B. Robic, and T. Ungerer, “Asynchrony in parallel computing: From dataflow to multithreading,” *Journal of Parallel and Distributed Computing Practices*, vol. 1, no. 1, pp. 3–30, 1998.

[92] J. Silc, B. Robic, and T. Ungerer, *Processor architecture: from dataflow to superscalar and beyond.* Springer Science & Business Media, 2012.

[93] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, “Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture,” in *Proceedings of International Symposium on Computer Architecture (ISCA).* San Diego CA, USA: IEEE, 2003, pp. 422–433.

[94] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, “Tartan: evaluating spatial computation for whole program execution,” *ACM SIGARCH Computer Architecture News*, vol. 34, no. 5, pp. 163–174, 2006.

[95] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, “Dynamically specialized datapaths for energy efficient computing,” in *Proceedings of International Symposium on High Performance Computer Architecture (HPCA).* San Antoni, USA: IEEE, 2011, pp. 503–514.

[96] R. S. Nikhil and G. M. Papadopoulos, “T: A multithreaded massively parallel architecture,” *ACM SIGARCH Computer Architecture News*, vol. 20, no. 2, pp. 156–167, 1992.

[97] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. Voneicken, “TAM-a compiler controlled threaded abstract machine,” *Journal of Parallel and Distributed Computing*, vol. 18, no. 3, pp. 347–370, 1993.

[98] J. Strohschneider and K. Waldschmidt, “Adarc: A fine grain dataflow architecture with associative communication network,” in *Proceedings of Euromicro Conference. System Architecture and Integration.* Liverpool,UK: IEEE, 1994, pp. 445–450.

[99] H. H. Hum, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, G. R. Gao, P. Cupryk, N. Elmasri, L. J. Hendren, A. Jimenez *et al.*, “A design study of the EARTH multiprocessor,” in *Proceedings of Parallel Architecture and Compilation Techniques (PACT)*, vol. 95. St. Petersburg, Russia: Citeseer, 1995, pp. 59–68.

[100] G. M. Papadopoulos and K. R. Traub, “Multithreading: A revisionist view of dataflow architectures,” in *Proceedings of International Symposium on Computer Architecture (ISCA)*, Toronto Ontario, Canada, 1991, pp. 342–351.

[101] L. Roh and W. A. Najjar, "Design of storage hierarchy in multithreaded architectures," in *Proceedings of IEEE International Symposium on Microarchitecture.* Michigan, USA: IEEE, 1995, pp. 271–278.

[102] K. M. Kavi, R. Giorgi, and J. Arul, "Scheduled dataflow: Execution paradigm, architecture, and performance evaluation," *IEEE Trans. Computers*, vol. 50, no. 8, pp. 834–846, Aug. 2001. [Online]. Available: `http://www.dii.unisi.it/ giorgi/papers/Kavi01a.pdf`

[103] C. Kyriacou, P. Evripidou, and P. Trancoso, "Data-driven multithreading using conventional microprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 10, pp. 1176–1188, 2006.

[104] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 162–173. [Online]. Available: `http://doi.acm.org/10.1145/1250662.1250683`

[105] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero, "Task superscalar: An out-of-order task pipeline," in *Proceedings of IEEE International Symposium on Microarchitecture.* Atlanta,USA: IEEE, 2010, pp. 89–100.

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

[107] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts essentials.* John Wiley & Sons, Inc., 2014.

[108] F. Petrini, D. J. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q," in *Proceedings of the ACM Conference on Supercomputing*, Phoenix, AZ, USA, 2003.

[109] P. Terry, A. Shan, and P. Huttunen, "Improving Application Performance on HPC Systems with Process Synchronization," *Linux J.*, vol. 2004, no. 127, 2004.

[110] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts, "Improving the scalability of parallel jobs by adding parallel awareness to the operating system," in *Proceedings of the ACM Conference on Supercomputing*, Phoenix, AZ, USA, 2003.

[111] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. Badia, E. Ayguade, and J. Labarta, "Productive cluster programming with OmpSs." Bordeaux, France: Springer, 2011, pp. 555–566.

[112] Y. Wu, L. Zheng, B. Heilig, and G. R. Gao, "HAMR: A dataflow-based real-time in-memory cluster computing engine," *The International Journal of High Performance Computing Applications*, vol. 31, no. 5, pp. 361–374, 2017.

[113] K. Kavi, J. Arul, and R. Giorgi, "Performance Evaluation of a Non-Blocking Multithreaded Architecture for Embedded, Real-Time and DSP Applications," in *Proceedings of International Conference on Parallel and Distributed Computing Systems (ISCA-PDCS-01)*, Richardson, TX, USA, 2001, pp. 365–371.

[114] R. Giorgi and Z. Popovic, "Core Design and Scalability of Tiled SDF Architecture," in *HiPEAC ACACES-2006*, L'Aquila, Italy, July 2006, pp. 145–148, poster. [Online]. Available: `http://www.dii.unisi.it/ giorgi/papers/Giorgi06a.pdf`

[115] R. Giorgi, F. Khalili, and M. Procaccini, "An FPGA-based Scalable Hardware Scheduler for Data-Flow Models," in *Proceedings of International workshop on FPGAS for Domain Experts (FPODE)*, Limassol, Cyprus, 2018, pp. 1–1, poster.

[116] F. Khalili, M. Procaccini, and R. Giorgi, "Reconfigurable logic interface architecture for cpu-fpga accelerators," in *HiPEAC ACACES-2018*, Fiuggi, Italy, 2018, pp. 1–4, poster. [Online]. Available: `http://www.dii.unisi.it/ giorgi/papers/Khalili18-acaces.pdf`

[117] V. A. Lorentzos, *Efficient network interface design for low cost distributed systems.* Master thesis at Technical University of Crete, 2017. [Online]. Available: `http://purl.tuc.gr/dl/dias/9F0A3578-6759-426E-A687-0B126EA3F684`

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

[119] A. Rizzo, G. Burresi, F. Montefoschi, M. Caporali, and R. Giorgi, "Making IoT with UDOO," *Interaction Design and Architecture(s)*, vol. 1, no. 30, pp. 95–112, 2016.

[120] EVIDENCE, s.r.l., "http://www.evidence.eu.com/."

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