HABILITATION À DIRIGER DES RECHERCHES

Délivrée par l'Université de Rennes
Spécialité : *Traitement du Signal et de l'Image*

Design Automation and Multi-Objective Optimizations for High-Performance Embedded Software

Par
Karol DESNOS

Présentée et soutenue à l'INSA Rennes, le 25.06.2024
Unité de recherche : Institut d'Électronique et des Technologies du numéRique (IETR)

Rapporteu·r·se·s avant soutenance :

Abdoulaye GAMATIÉ	Directeur de Recherche CNRS, LIRMM, Montpellier
Tanguy RISSET	Professeur des Universités, CITI, INSA Lyon, Villeurbanne
Marilyn WOLF	Professor, University of Nebraska, Lincoln, US

Composition du Jury :

Président :	Guy GOGNIAT	Professeur des Universités, Lab-STICC, Université Bretagne Sud, Lorient
Examinateurs :	Dominique GINHAC	Professeur des Universités, ICB, Université de Bourgogne, Dijon
Garant d'HDR :	Christophe MOY	Professeur des Universités, IETR, Université de Rennes, Rennes

Oh wow! Much thanks, very grateful, so recognizing, very appreciative, how acknowledging.

Acknowledgments

As we all know, this section is likely the one with the highest chance of being read after the habilitation defense. For this reason, I apologize in advance for the length of this section, but I have at heart to take this opportunity to thank all the people with whom I have had the privilege of working with (and living with) over the past 10 years.

First of all, I'd like to thank all the members of my habilitation jury, starting with Abdoulaye GAMATIÉ, Tanguy RISSET, and Marilyn WOLF. Thank you for taking the time to read this manuscript, and thank you for your insightful reports. Thanks also to Guy GOGNIAT and Dominique GINHAC for participating in the defense, and for taking the risk of being appointed by the university as a reviewer. Finally, special thanks to Christophe MOY for accepting the role of guarantor for this habilitation. Thank you for guiding me through the university administrative process, for helping me find the jury, and for proofreading the manuscript before submission. Thanks to all members of the jury for their participation in the defense and for the stimulating scientific discussions thereafter.

This work would not be possible without all the colleagues of the VAADER research team. In particular, I would like to express my gratitude to all the Ph.D. students: Julien HASCOËT, Hamza DEROUI, Justine BONNOT, Alexandre HONORAT, Florian ARRESTIER, Alban MARIE, Mewe-Hezoudah KAHANAM, Ophélie RENAUD, Alice CHILLET, Quentin VACHER and Paul ALLAIRE. I must humbly say that this work is yours before it is mine, and thank you for all that I have learned and am learning as I do my best to accompany you on the bumpy road of your doctoral studies. Thanks also to all the VAADER's directors and co-supervisors for these PhDs, with whom it was a pleasure to work closely, even when it meant censoring wildly out-of-control brainstorming sessions: Daniel MENARD, Jean-François NEZAN, Maxime PELCAT, Mickaël DARDAILLON, Nicolas BEUVE, Thibaut MARTY, Luce MORIN, and Lu ZHANG. Thanks also to the extra-VAADER co-supervisors for these PhDs: Alix MUNIER-KORDON, Eduardo JUAREZ, Matthieu GAUTIER, Robin GERZAGUET, and Guillaume FORTIER. Many thanks also to the research engineers, postdocs, or graduate students in the team with whom I have worked over the past 10 years: Clément Guy, Antoine Morvan, Alexandre Mercat, Nicolas Sourbier, Julien Heulot, and Hugo Miomandre. Although I cannot name them all for fear of adding another 100 pages to this manuscript, I would like to thank all the interns, masters, and InnovR students for their hard work, as well as the other members of the Vaader team for the stimulating work environment.

Many people from INSA Rennes and the IETR laboratory also deserve their share of gratitude. Although not strongly reflected in this manuscript, teaching collaborations should not be forgotten, especially with Muriel Pressigout, Marie Babel, Bruno Arnaldi, and François Pasteau. No collaborative project, conference participation, PhD defense, or daily work would be possible without the administrative support of Corinne Calo, Jocelyne Tremier, Isabelle Eveillard, Christèle Duhoux, Patricia Soufflet, Cécile Rocuet and Pierre Corvec of the 2PE platform, as well as the INSA Research Support Office and INSA services in general. Special thanks also to Frédéric Garesché, Laurent Guillaume and Jérémy Dossin for their unfailing IT support.

Over the past 10 years, I have had the opportunity to work with many wonderful French colleagues, both from academia and industry: Kevin Martin, Nicolas Gac, Andrea Enrici, Jocelyn Sérot, Sébastien Le Nours, Ruben Salvador, Slaheddine Aridhi, Olivier Gesny, Erwan Raffin, and Erwan Nogues. One of the benefits of being a researcher is that you get to meet amazing people who work all over the world, including: Francesca Palumbo from UNISS, the tireless UPM teams with Fernando Pescador, Miguel Chavarrías, Cesar Sanz, Eduardo de la Torre, and Alfonso Rodriguez; Jeronimo Castrillon from TUD, Johan Lilius from Åbo Akademi, John McAllister from QUB, Tomasz Kryjak and Hubert Szolc from AGH, Stephen Kelly from McMaster University, and Adam Deller from Swinburne University. This paragraph would not be complete without extending a few thanks to (most of) the partners of the Cerbero project, and especially to the great graduate students who brought life to this EU project: Raquel Lazcano-Lopez, Daniel Madroñal, Tiziana Fanni, Claudio Rubattu and Leonardo Suriano.

A few additional words of thanks to the unknowing contributors to this manuscript. Thanks to Deepl Write for advising on the style of this manuscript. Thanks to my physiotherapist Aurore Garrau-Gauzins for helping me realign my vertebrae in order to finish writing this manuscript. Thanks to my colleagues who participate in our Friday board game sessions for backstabbing me so I can get back to writing faster. And thanks to the organizers of HellFest Open Air for making sure I got my annual dose of intravenous heavy metal fun.

To conclude this section, I would like to thank my primary supporters for the past 36 years: my parents, grandparents, sister and brother. Thank you for your love, thank you for always being there and for raising me. Thanks also to my in-laws for welcoming me and making me a part of the Martial family.

Thanks to the three little monsters, Loreleï, Azaël and Maëline, for stealing 90% of my free time. Thank you for bringing so much love, fun, joy and sleep deprivation into my life.

Last but exact opposite of least, thank you Pauline for sharing my life for the last `<insert-valid-count>` years. I'm sorry I didn't get you a ring the day of this defense, I hope you'll forgive me someday. Thank you for enduring me, thank you for gifting me with these three wonderful children, thank you for sharing all the good (and bad) moments by my side, thank you for being you, always. I could not have hoped for a better partner in life.

Contents

Pika Pika! Piiiikaaaachuuuuu!

Pikachu, Pallet town

Chapter 1

Introduction: Research Activities Context and History

The present habilitation (*French:* Habilitation à Diriger des Recherches (HDR)) manuscript synthesizes the research objectives pursued in the 10 years following my PhD defense, back in fall 2014. The research presented in this manuscript covers the work achieved in the three positions held during this period of time:

- 09.2014 - 08.2015: Assistant Professor (*French:* Attaché Temporaire d'Enseignement et de Recherche (ATER)) at INSA Rennes,
- 09.2015 - 11.2015: Research Engineer at INSA Rennes,
- Since 12.2015: Associate Professor (*French:* Maître de Conférences) at INSA Rennes.

All three positions were held with a joint research appointment in the IETR (Institut d'Electronique et des Technologies du numéRique) laboratory, UMR6164 of the CNRS, as a member of the VAADER (Video Analysis and Architecture Design for Embedded Resources) research team.

This chapter is organized as follows: Section 1.1 introduces the scientific and technological context of this manuscript, and Section 1.2 gives an overview of the scientific challenges tackled in my work. Then, as research work generally is the result of collaborative efforts, Section 1.3 gives an overview of my research collaborations, covering the history of PhD supervisions and participation to research projects.

1.1 Embedded System Design Context

1.1.1 Embedded systems

The design and optimization of high-performance embedded computing systems is a complex task which constantly requires pushing back the frontiers of scientific and technological knowledge in many domains. By definition, an embedded computing system is the combination of a computing infrastructure, called *hardware*, which executes computations of a given application, called *software*.

Embedded systems are generally dedicated to providing a well-specified service with a great efficiency, as opposed to general-purpose computing systems that can serve many purposes, but with poor efficiency. Hence, the design of embedded system is a highly complex task due to the duality between hardware and software concerns, and due to the tight constraints imposed by the service to be provided. In addition to this intrinsic design complexity, new technologies, societal needs, and innovations constantly drive the need for new design techniques, both for hardware and software, and require the consideration of new design constraints.

The hardware and software stack

This simplistic dual view of a computing system can be supplemented and refined with several other layers to form a so-called stack. Figure 1.1 illustrates the various layers of the complete hardware and software stack considered in this work. The basic principles of each layer are detailed hereafter:

- **Domain-specific layer:** At the top of the stack, the highest level of abstraction provides domain-specific languages and tools to developers. This high-level layer takes the form of Application Programming Interfaces (APIs), which provide developers with software primitives for interacting with the underlying layers. These APIs are typically associated with highly optimized full-stack implementations, allowing developers to take full advantage of the computing capabilities of the system. The main advantage of these APIs is that they can be used by domain experts who may have limited knowledge of the underlying layers of the system stack. Examples of such domain-specific APIs include *TensorFlow* for deep-learning algorithms [2], *DALiuGE* for radio-astronomy applications [108], and *GNU Radio* for software defined radio applications [10].
- **Generalist Programming Methods:** The main layer of the software part of the stack gathers languages, models, and tools that can be used to program and optimize

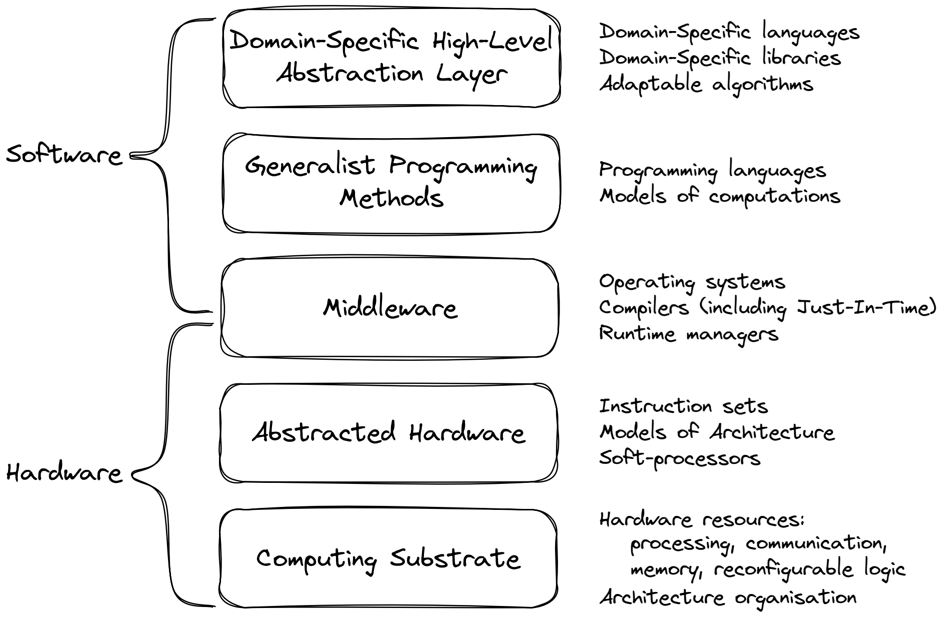

Software
Hardware
Domain-Specific High-Level Abstraction Layer
Generalist Programming Methods
Middleware
Abstracted Hardware
Computing Substrate
Domain-Specific languages
Domain-Specific libraries
Adaptable algorithms
Programming languages
Models of computations
Operating systems
Compilers (including Just-In-Time)
Runtime managers
Instruction sets
Models of Architecture
Soft-processors
Hardware resources:
processing, communication,
memory, reconfigurable logic
Architecture organisation

Figure 1.1: A personal view of the hardware and software stack

many types of systems and applications. Models of Computation (MoCs) [88] formalize the mathematically-grounded semantics that can be used to describe the behavior of a system. Based on this model-based description, key properties of a system can be guaranteed, either by construction or by exploiting the mathematical formalism associated with the model used. For example, a MoC may help guarantee the absence of deadlock while supporting parallel computation, or may enable the specification and verification of real-time properties. Programming languages implement one or several MoCs, which they expose to the programmer through their syntax and grammar. Although MoCs are hardware-agnostic, programming languages are generally dedicated to a certain type of hardware, through a compiler tool-chain. For example, *Python* is used for the programming of General Purpose Processorss (GPPs), while *Verilog* is used for the description of hardware. However both languages may implement the finite-state machine MoC.

- **Middleware:** At the frontier between hardware and software, the middleware layer provides necessary services for the deployment of software on hardware. On the software side, the middleware layer offers low-level services that relieve upper layers of the software stack from hardware concerns. For instance, operating systems or runtime managers automatically schedule and map the software threads on available cores of the architecture. On the hardware side, certain software may require additional hardware components to be deployed on a computing architecture. For example, memory management units are necessary for deploying complex operating systems like *Linux*.

- **Abstracted Hardware:** When co-designing an embedded system, manipulating high-level abstractions of the hardware and software components eases the Design Space Exploration (DSE). Therefore, high-level hardware models are used to capture key properties and characteristics of the hardware without the complexity needed to actually build the hardware. For example, coarse Models of Architecture (MoAs) can be used to describe an architecture and predict with a high fidelity the energy consumption of deployed software [J12]. Another example of hardware abstraction is the description of a processor through its instruction set, which suffices to compile a piece of software into optimized assembly code.
- **Computing substrate:** The actual hardware components used to build an architecture constitute the lower level of the stack considered in this work. Each processing, memory, and communication component integrated within a computing substrate is by itself a whole research domain, well beyond the scope of this manuscript.

The stack presented in Figure 1.1 is a personal view of the hardware and software layers, inspired by research works covered in this manuscript. Other views of the hardware and software stack, influenced by subjective system needs, can be found in the literature [14]

1.1.2 Trends in Embedded Computing Systems

Hardware trends

In the past half-century, the ever-increasing complexity and capabilities of computing hardware has been closely linked to the number of transistors that can be integrated into a single chip. Indeed, this fabulous increase of hardware complexity is made possible by the exponential increase in the achievable density of transistors, dictated by Moore's law [67]. While some researchers believe that, due to physical constraints, Moore's law will soon come to an end, *TSMC*, one of the world's leading semiconductor foundries, still argues that Moore's law will not end in the near future [107], thanks to upcoming innovations in nanotechnologies.

One of the most impacting recent trends in computing architecture, although it is not the most technically innovative, is the rise of open-hardware initiatives, notably with the efforts around the RISC-V project [103]. RISC-V (Reduced Instruction Set Computer

(RISC) five) is an open and extensible Instruction Set Architecture (ISA), originally introduced by *UC Berkeley (USA)* in 2010, and whose development is now overseen by the non-profit *RISC-V foundation (CH)*. An ISA defines a set of assembly instructions that must be supported by any computing architecture claiming compatibility with this ISA. Although a standard ISA, such as RISC-V, does not specify or provide any reference implementation supporting the instruction set, many open-source implementations can be found in the scientific literature [32, 106]. Several commercial chips based on the RISC-V architectures are already available, notably low-power Digital Signal Processing (DSP) chips from *GreenWaves Technologies (FR)*, or high-performance cores from *SiFive (USA)*. For academics, the RISC-V architecture is a fantastic opportunity to experiment new architectural concepts with an extensible ISA, while benefiting from a rich and well-maintained eco-system, including dedicated design tools [49], ISA simulators [84], or state-of-the-art compilers [78].

Among trending concepts in the architecture domain, in-memory [90] and near-memory [93] computing have received a lot of attention in recent years. The motivation behind these concepts is the well-known *memory wall* issue [109], which makes data accesses and movements a performance and energy bottleneck in modern computing systems. The cause of this memory wall is that memory technologies do not keep up with the rapidity of access and capacity needs imposed to them by processing elements and data-intensive algorithms. To limit costly data movements, and circumvent the memory wall, the strategy adopted by in- and near-memory computing is to move computations directly within memory banks, using memristors or charge-based systems; or on the physical edge of memory banks using small Processing Elements (PEs) with limited capabilities, that receive instructions from the host Core Processing Unit (CPU) of the architecture. Preliminary performance evaluation with such architectures show promising results, notably in data-intensive domains such as Artificial Intelligence (AI) [90]. To take full advantage of the power-efficiency offered by in- and near-memory computing hardware, should it become a commercial standard, compilers and other algorithmic optimization tools will need to be adapted.

The last decades has also witnessed many evolutions in the computer architecture domain:

- Multiprocessor Systems-on-Chips (MPSoCs) integrating both GPPs and reconfigurable logic, first commercialized in 2011 by *Xilinx (USA)*, now *AMD (USA)*, [81], have become more mainstream architectures, notably available in major cloud-computing services.

- Use of Graphics Processing Units (GPUs), and the evolution of their architectures, to better support general purpose computations is another major event in the architecture domain. Since 2012, the GPU market is boosted by the development of deep-learning techniques [53].

- New domain-specific architectures have been introduced, such as Tensor Processing Units (TPUs) [42] for accelerating AI computations.

- The number of cores integrated in processors has steadily increased. For example in the Xeon family of processors from *Intel (USA)*, the maximum number of physical core has increased from 8 cores in 2010, up to 56 cores in 2019 [39]. Another example of many-core architecture, is the third generation chip developed by *Kalray (FR)*, which embeds 80 cores [21].

- Techniques and tools used for hardware design have also evolved in the last decade, with a wider adoption of High-Level Synthesis (HLS) techniques, making hardware design possible from high-level programming language, such as C++. A striking example of this adoption is the use of HLS by *Google (USA)* to develop a hardware video encoder for their Youtube platform [54].

Software and application trends

On the software side, widespread adoption of new concepts, in the form of new programming languages or tools, is paradoxically much slower than in hardware. For example, among the 10 most popular general purpose programming languages ranked every year by *IEEE* [95], all 10 languages have been created, or are heavily based on a language created more than 20 years ago. `Rust`, a trending language created in 2006 favoring safe and reliable concurrent programming, is ranked at the 20th position in terms of popularity. Unsurprisingly, `C` and `C++` monopolize the 2nd and 3rd positions in this ranking, and probably take the lead among embedded systems programmers, as they already did in 2009 [5].

While technologies used to implement embedded software do not evolve rapidly, despite being a hot research area (see Section 1.2.2), applications and services supported by embedded systems evolve rapidly.

On the embedded devices side, IoT devices have become extremely popular in the last few years, backed by the rapid evolution of telecommunication standards [15]. Internet-of-Things (IoT) devices are small embedded computing systems which generally embed sensors, and sometimes actuators, and that communicate with other devices or with a master system through a wireless internet connection. The pervasive use of these computing devices opens the way to many new applications, including: industry 4.0, smart agriculture, smart cities, smart grids, autonomous vehicles, and e-Health, to cite only a few. Each of these application domains comes with its own set of challenges and constraints for the development of embedded software.

New generations of applications also need to be developed, pushed by the creation of new technologies, by scientific needs, or by the evolution of standards in different domains. For example, giant international scientific infrastructures such as the CERN particle accelerator or the SKA radio-telescope, constantly require additional processing capabilities to handle the gigantic amount of data they generate. The evolution of telecommunication and image compression standards is another example of constantly evolving application needs where the software of each new generation is often many times more complex than the previous, following the infamous Wirth's law [105].

In addition to the apparition of new applications, new algorithms and data-processing techniques are invented or refined with a sustained pace. Many existing systems need to be updated to integrate these new algorithms and benefit from their advantages. A striking example of such evolution is the recent advent of deep-learning algorithms. In less than ten years, since the breakthrough of AlexNet in the ImageNet challenge [53], the use of deep-learning algorithms has skyrocketed in all application domains.

Design constraints trends

In recent years, the awareness of new constraints for designing embedded systems has spread rapidly, for all applications domains and on all types of hardware.

While reliability and safety constraints have always been at the core of many embedded system designs, cybersecurity has recently become a major concern for most systems. A primary reason for this concern is that the ubiquitous use of vulnerable interconnected embedded devices opens the door to attacks with catastrophic consequences. The potential threats posed by vulnerable devices range from privacy issues, where personal or corporate data can be stolen, to life-threatening issues, where, for example, a pacemaker can be remotely disabled [79]. Although the connectivity features of embedded devices are a major entry point for attackers, other techniques can be used, such as hardware attacks that exploit physical and logical weaknesses to leak sensitive data or take control of a system.

Another important recent constraint for the design of modern embedded systems is the consideration of sustainability constraints. Minimizing the power consumption of embedded systems has been the subject of much research, most often motivated by the need to extend the battery life of a system. While reducing the energy consumption of systems may have an undeniable impact on their carbon footprint, the design of sustainable embedded systems goes far beyond this goal and considers the entire life cycle of the system. Designing sustainable systems requires addressing complex issues such as sourcing electronic components, designing maintainable, reusable, and recyclable systems, and optimizing their power consumption at all stages of their lifecycle [75].

Within this context, during the first years of my career, my research work has been focused on design automation and optimization for low-level software on high performance embedded systems.

1.2 From Multi-core Programming to Many-Facet System-of-Systems Optimization

1.2.1 Career Starting Point: Focus of the PhD Thesis

The PhD work defended in 2014 focused on the programming of Multiprocessor Systems-on-Chips (MPSoCs) based on dataflow Models of Computation (MoCs) [B4].

The first set of contributions from this thesis focused on the optimization of the memory footprint allocated to run applications specified with Synchronous Dataflow (SDF) graphs [59]. In particular, contributions introduced a graph modeling the constraints that condition the validity of an allocation of memory used to transfer data between concurrent tasks exposed by the dataflow model. Based on this model, the memory footprint to be allocated could be bounded [J5, C20], statically allocated on shared memory [J5, J6, C18, C21] and distributed memory architectures [C19].

The second contribution of this thesis was an extension of the SDF MoC. The main advantage of using an SDF graph to describe the behavior of an application is that it captures the task, data, and pipeline parallelism of applications while guaranteeing determinism and the absence of deadlock at compile time. In addition to existing features, the Parameterized and Interfaced SDF (PISDF) extension of the SDF MoC brings new hierarchy and reconfigurability features to the model [C22]. Compositional hierarchy allows an application to be decomposed into separately analyzable subgraphs, while reconfigurability allows the modeling of applications with more data-dependent dynamic behavior than in the SDF MoC [C27].

1.2.2 Tackled Research Challenges

Since the end of my PhD, the scope of my research has broadened and is now focused on the **multi-criteria optimization of middle-ware for the design of embedded systems.** This research area is at the intersection of several research areas, and most importantly: Computer Aided Design (CAD) techniques for MPSoCs programming, operational research, compilation, image and signal processing, computer vision, and artificial intelligence.

The following paragraphs summarize the main challenges tackled in my research between 2014 and 2024. It should be noted that the purpose of this introductory section is not to summarize all the contributions proposed during this work period, but to give a global scope of the scientific challenges motivating the work detailed in the following chapters.

Design levers for embedded system optimization

The co-design flow for embedded systems is a complex process involving many steps at different levels of abstraction, many models, and often many tools. At each step of the design process, many decisions can be made, each of which has an impact on the performance and quality of the built system. An important challenge for a researcher in the field of CAD is to expose and exploit the different design levers, at all steps of the design process, that allow developers to seamlessly optimize their embedded system.

Algorithmic choices are a first lever to optimize a system. There are often many different algorithms available to solve a given problem. Some algorithms may be inherently different, each providing a different solution to a problem, with a different approach, and each having its own advantages and drawbacks. For example, 184 algorithms are evaluated in [100] for the stereo matching problem, which consists of reconstructing the depth of a scene from two images captured by a pair of cameras. Each of these 184 algorithms offers a different trade-off between accuracy and suitability for real-time implementation on different types of hardware targets. Once a well-defined algorithm has been chosen, algorithmic choices can still be made about how this algorithm is implemented. For example, by using approximate computation techniques, it is possible to trade off the accuracy of the algorithm for a reduction in memory space [C7, 66], or for a reduction in computational complexity [C8]. Different implementations of an algorithm may also be functionally equivalent, but exhibit different non-functional properties, such as their degree of parallelism [C28], or their task granularity [C43].

Resource allocation choices are another key lever for optimizing an implementation. Resource allocation consists of the logical, temporal, and spatial distribution of the hardware resources required to support the execution of a system. Resources to be allocated typically include: processing elements, memory, communications, and reconfigurable logic.

- Processing elements: The computations of the system must be logically and temporally allocated, that is mapped and scheduled, to the available processing elements of the architecture. This allocation is often highly complex due to the presence of heterogeneous processing elements with different capabilities [C23, C27]. In addition to mapping and scheduling, managing processing elements also requires taking care of idle and sleep modes, and configuring the speed and energy tradeoff of the processing element using DVFS (Dynamic Voltage and Frequency Scaling) [C28, C29, C45].

- Memory: Data received, processed, and produced by a system must be stored in the memory banks of the embedded architecture. In most embedded systems, memory is a scarce resource whose access is often a bottleneck for system computation [90]. The allocation of data in memory, both spatially and temporally, is therefore a key lever to optimize the performance of a system [J6, C21]. In addition to the scarcity of resources, the memory allocation problem is made more complex in the presence of complex memory hierarchies with distributed memory banks, caches and scratchpads [J5, C19].

- Means of communication: Data exchange and synchronization between processing elements must be scheduled on appropriate hardware to support the execution of an application. Different communication strategies can be used, such as synchronous and asynchronous communications [C26, 35], with active or passive wait mechanisms.

- Reconfigurable Logic: Major semiconductor companies, namely *Intel* and *AMD*, now offer MPSoCs that integrate both classical processing elements (GPPs, GPUs) and reconfigurable logic. To take advantage of this hardware, the reconfigurable logic must be spatially configured with a bitstream, either statically at compile time or at runtime with dynamic partial reconfiguration [J15].

Optimization strategies

There are many different strategies, at all stages of the design process, to exploit the aforementioned optimization levers [57].

Early in the design process, choosing the right algorithm and modeling it with the right MoC [85] is a major lever to guarantee key properties of a system by design, such as real-time properties [C29].

The largest corpus of work on exploiting these optimization levers focuses on compilation steps. Compilation steps encompass many different types of optimizations, including model-based optimizations and code generation [J9, C17, C43, C45], source-to-source transformations [6], resource allocation [C19, C23], and classical code compilation [B2].

Some optimization levers can also be exploited while the system is running [C21, C27]. While dynamic optimizations provide great adaptability to the system, they require the use of a runtime manager or operating system, which imposes an overhead on the performance and resource usage of the system [J3]. Another drawback of dynamic optimizations is that while they may be more efficient than their static counterparts [C21], it is sometimes impossible to give guarantees on their behavior, which may be a problem in systems with hard real-time or tight hardware resource constraints [C39].

There also exist hybrid strategies that rely on both compile-time and run-time optimizations. For example, reconfigurable dataflow semantics MoCs [B3] supports both compile-time analysis and optimization of the system, but requires a lightweight runtime manager to handle on-the-fly resource allocation of data-dependent computations. Other hybrid strategies, called quasi-static, consist of precomputing a set of resource allocations at compile time and switching between these precomputed allocations at runtime [26].

When designing a system, choosing the right combination of optimization techniques is inherently challenging. First, the impact of each individual optimization is often hard to predict and competing techniques each have their own strengths and weaknesses. Second, when multiple optimizations are chained together in a development flow, the interactions between different techniques can be difficult to predict. For example, a transformation applied early in the design flow may hinder the efficiency of a later one [J6, C43].

Multi-criteria optimization

In past works on embedded system design, optimization processes typically focused on a single optimization criterion. For example, many papers on mapping and scheduling strategies for dataflow graphs focused solely on minimizing the latency [29] or minimizing the memory footprint [28] of the system. Often, even when multiple optimization criteria are studied, one criterion is secondary, observed only as a by-product, or used as a constraint for optimizing a primary criterion [97].

In modern CAD techniques, system designers must consider and jointly optimize many different optimization criteria [C28]. The performance of the system, measured in terms of latency and throughput, its power and energy consumption, the quality of service provided, the utilization of hardware resources, such as memory, processing elements, or reconfigurable logic, are just a few examples of metrics to optimize. Unfortunately, optimizing one metric often has negative side effects on the value of another one, and acceptable trade-offs must be selected by the system designer. This set of acceptable solutions is usually represented as a front of Pareto efficient solutions. In a multicriteria optimization problem, a solution is said to be Pareto efficient if there is no other solution that improves the value of one criterion without sacrificing another.

The global design space to be explored for embedded system development is often extremely large, and it is not uncommon to have thousands or millions of potential configurations to evaluate. Since it is not feasible to evaluate the optimized metrics for all configurations of the system, alternative methods must be used to identify the Pareto-efficient configurations without relying on exhaustive exploration. The two main methods to drastically reduce the explored design space are to rely on an analytical approach or to rely on heuristics. The analytical approach consists of identifying the mathematical model that relates the configuration parameters to the observed metrics [97], which requires formal models for the system specification and for the effect of optimizations and processes in the design flow. Alternatively, a heuristic approach generally considers the effect of configuration parameters on optimized criteria as a black box process [C28]. Heuristic approaches navigate the design space with a wide range of strategies, such as gradient descent or genetic approaches, with no guarantee to identify the optimal set of Pareto-efficient configurations [C9].

Multi-criteria optimization problems have notably been studied in the following personal works: [C28, C30, C38, C43].

Embedded system hardware and software jungle

A final challenge for the development of embedded systems is the sustained pace at which new types of hardware and applications have emerged over the last 10 years.

On the hardware side, as was mentioned in Section 1.1.2, the use of heterogeneous MPSoCs has exploded in recent years, integrating GPPs [C15], GPUs, dedicated hardware accelerators [C47], reconfigurable logic [B3], and microcontrollers [C17] on a single chip. The increasing use of systems of systems for cyber-physical applications [23] also makes hardware considerations more complex for system designers. Indeed, the design of systems of systems that rely on interconnected and heterogeneous devices to provide a service requires the use of new design techniques inspired by the HPC (High-Performance Computing) literature, mixed with innovative approaches such as swarm intelligence.

On the application side, design complexity stems from the wide variety of application domains that must be considered, each with its own set of challenges, tools, and constraints. For example, the design domains considered in this thesis include stream processing and computer vision applications [J5, J8, C8], image and video coding [J11, C38], AI [C23, C37], cybersecurity [J14], and real-time systems [C29].

1.3 Research Activities History: Research Projects and Supervised Students

An overview of my research career, including supervised PhDs, collaborative research projects, and teaching activities is given in Figure 1.2. The topics of the supervised PhD theses are detailed in Section 1.3.1 and details on the collaborative projects are given in Section 1.3.2.

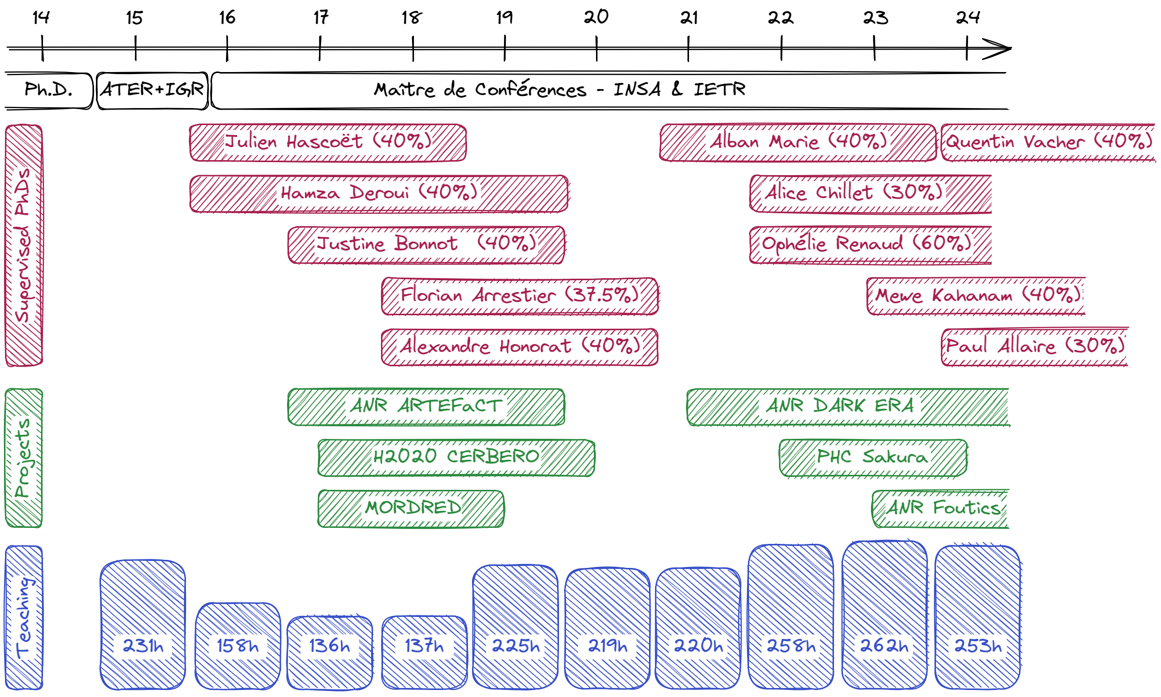


Figure 1.2: Research career at a glance

1.3.1 Supervised PhDs

A detailed view of supervised PhD students is presented in Table 1.1. For each student, this table gives the supervision rate, the names of the co-supervisors, the source of funding, and the date of the PhD.

	PhD Student	Superv. rate	Co-supervisors	Funding	Date
Defended PhDs	J. Hascoet	40%	J. Nezan (60%)	CIFRE Kalray	10.2015→9.2018
	H. Deroui	40%	J. Nezan (30%) A. Munier (30%)	MESR	10.2015→9.2019
	J. Bonnot	40%	D. Menard (60%)	ANR ARTEFACT	10.2016→9.2019
	F. Arrestier	37.5%	D. Menard (37.5%) E. Juarez (25%)	ANR ARTEFACT & H2020 CERBERO	10.2017→9.2020
	A. Honorat	40%	J. Nezan (60%)	H2020 CERBERO & Regional Funds	10.2017→9.2020
	A. Marie	40%	L. Zhang (30%) L. Morin (30%)	Regional Funds & INSA	10.2020→9.2023
PhDs in Progress	O. Renaud	60%	J. Nezan (40%)	ANR DARK ERA	10.2021→9.2024
	A. Chillet	30%	R. Gerzaguet (40%) M. Gauthier (30%)	DGA	10.2021→9.2024
	M. Kahanam	40%	M. Pelcat (30%) G. Fortier (30%)	CIFRE Inetum	12.2022→10.2025
	Q. Vacher	40%	M. Dardaillon (30%) T. Marty (30%)	ANR FOUTICS	10.2023→9.2027
	P. Allaire	30%	M. Dardaillon (40%) N. Beuve (30%)	ANR FOUTICS	10.2023→9.2027

Table 1.1: Details of supervised PhDs.

In continuity with my PhD, most of the supervised PhDs focus on the use of dataflow MoCs for programming heterogeneous MPSoCs. In particular, the PhDs of Julien HASCOËT and Ophélie RENAUD focus on controlling the granularity of dataflow graphs to minimize the complexity of resource allocation while maintaining system performance [C25, C43]. Hamza DEROUI's PhD focuses on developing fast analytical heuristics to predict the latency and throughput of hierarchical dataflow graphs deployed on multicore targets [C14, C15]. Florian ARRESTIER studied extensions to dataflow MoCs for specifying persistent data [C5] and developed a numerical representation that replaces memory- and time-consuming graph transformations used for resource allocation of reconfigurable dataflow graphs [J3]. Alexandre HONORAT developed dataflow modeling and optimization techniques for multi-objective resource allocation under real-time constraints [C28–C30].

Justine Bonnot and Alban Marie's PhD research explores how approximations of computation and data can be used to build better systems. Justine Bonnot has developed techniques to rapidly simulate the error of hardware approximation operators [C6, C11], and models and heuristics to support and accelerate the design space exploration process for approximate computing systems [C9, P1]. Alban Marie works in the context of Video Coding for Machines (VCM). His goal is to explore the accuracy-bandwidth tradeoff made possible by the resilience of deep learning AIs to image artifacts caused by lossy compression.

Alice Chillet's and Mewe Kahanam's PhD research focuses on the development of low-power machine learning techniques. Alice Chillet is working on using deep learning and genetic programming techniques to identify Radio Frequency (RF) emitters by detecting their unique RF fingerprint caused by small variations in their manufacturing process. Mewe Kahanam's goal is to create mathematical models that predict *a priori* the benefits for different deep learning complexity reduction techniques, on different types of layers, and on different types of hardware.

Finally, Quentin Vacher's and Paul Allaire's PhD will study the design of ultra-low power Reinforcement Learning (RL) agents built with an innovative technique inspired by genetic programming.

1.3.2 Collaborative projects

A detailed overview of my personal involvement in research projects is presented in Table 1.2. This table shows the name of the projects, the source of funding, my role in the consortium, the name of the leading institution, and the total budget of the projects.

Date	Project	Funding	Personal role	Leader	Budget
2011-15	Compa	ANR	Partner	IETR/INSA	800k€
2016-19	ArteFacT	ANR	Participant	CEA Leti	660k€
2017-20	Cerbero	H2020	INSA scientific coord. & Task leader	IBM	5.3M€
2017-18	Mordred	GdR ISIS	Leader	IETR/INSA	6.5k€
2021-25	Dark Era	ANR	Participant	L2S	498k€
2022-23	Mobilité	PHC Sakura	Co-writer & Participant	INSA	19k€
2023-26	Foutics	ANR	Co-leader	INSA	282k€
2023-xx	Rising Stars	Horiz. Eur.	Participant	Obs. de Paris	634k€

Table 1.2: Details of funded collaborative projects.

Funded projects

Five of the collaborative projects in which I have actively participated study the use of dataflow MoCs for programming parallel architectures: Compa, Cerbero, Mordred, Dark Era, and Rising Stars. In Compa, reconfigurable dataflow MoCs and associated runtime managers were developed, as well as application graphs for standard video decoders [C27]. In Cerbero, a model-based tool chain integrating components from several partners was built, covering high-level application modeling with Preesm [C5, C41] (INSA), support for runtime adaptation in software with Spider [J3, J15, C27] (INSA) and in hardware with Artico3 (UPM-CEI) and MDC [73] (UniSS), detailed profiling with Papify [J10] (UPM-CITSEM), and on-the-fly polyhedral optimization with Apollo [56] (UPM-CITSEM & Inria). In Mordred, a *young researcher* project I personally led, the Spider runtime manager for reconfigurable dataflow graphs was ported to the *Kalray MPPA2*, a many-core architecture with 288 cores [C39]. Finally, the goal of the Dark Era and Rising Stars projects is to extend the use of the embedded dataflow MoCs to the HPC system in order to support exa-scale radio astronomy projects, such as the Square Kilometer Array (SKA) giant radio telescope.

The ArteFact project focused on the study and development of CAD methods based on approximate computation techniques. The PHC Sakura project supported the mobility of graduate students and researchers between Japan and France to initiate collaborative work on VCM.

The Foutics project is a young researcher project, co-proposed with Mickaël Dardaillon, with equal involvement, and officially submitted by M. Dardaillon. The goal of the Foutics project is to develop energy-efficient reinforcement learning agents based on the Tangled Program Graph (TPG) model, inspired by genetic programming techniques. A full-stack and energy-centric approach is considered in this project, from the integration of energy concerns in the TPG model to its energy-efficient implementation on low-power embedded devices and reconfigurable logic.

Non-funded projects

Writing project proposals is an important scientific activity for a researcher, but it is not always successful. Nevertheless, the fruitful scientific discussions with the partners of the written proposals and the important time spent on this activity justify, in my opinion, their appearance in the section related to research activities. Many projects that were not funded in previous attempts may succeed in the future, as was the case with the Cerbero, Dark Era, and Foutics projects, all of which were funded in their second attempt.

The list of non-funded collaborative projects I have contributed to is given in Table 1.3. Most of the submitted projects focus on the model-based programming of MPSoCs, including BusyStream, two follow-on projects from Cerbero: Heracles and Rubeus, Visygo and Gigantic. The goal of these projects was to study the design of cyber-physical systems of systems, swarm intelligence, and generative design techniques. The Licornn and Dacofab projects focus on the Video Coding for Machines (VCM) problem, which later led to the PhD thesis of Alban Marie. Finally, the goal of the aimed project was to exploit the simplicity and adaptability of TPGs in the context of explainable AI.

Project partners

Fruitful collaborations with the following institutions were established during these projects, but also during direct collaborations:

- **Companies**: Texas Instruments (Fr, USA), IBM (Isr), Kalray (Fr), Nokia Bell Labs (Fr), Abinsula (It), Atos (Fr), Silicom (Fr), Inetum (Fr)
- **Research Centers**: Inria (Fr), TNO (NL)
- **Academics**: Universidad Politecnica de Madrid (UPM, Es), Università degli Studi di Sassari (UniSS - It), Hosei University (Jap), University of Maryland (USA), Technische Universität Dresden (Ger), McMaster University (Can), Queen's University Belfast (UK), EPFL (CH), Abo Akademi (Fin), ENIT (Tun), ENIS (Tun), Irisa (Fr), Lab-STICC (Fr), Lip6 (Fr), L2S (Fr), Institut Pascal (Fr)

Attempts	Project	Funding	Personal role	Leader	Partners	Grade	Budget
2016	BusyStream	H2020	INSA Leader	Linköping U.	11	13/15	5.5M€
2019	Heracles	H2020	INSA Leader	Eurescom	10	10/15	4.9M€
2019	Dacofab	Cominlabs	Leader	INSA	2	-	179k€
19,20,21	Visygo	ANR/DFG	Partner	INSA	4	-	480k€
19,20,21	Licornn	ANR JCJC	Leader	INSA	1	B	247k€
2020	Aimed	Chist-era	Partner	Uni. Vaasa	3	-	470k€
2022	Rubeus	Horiz. Europe	Partner	Abinsula	13	11.5/15	7.8M€
21,22	Gigantic	Horiz. MSCA	Partner	INSA	13	88%, 74%	2,7M€

Table 1.3: Non-funded collaborative projects.

1.4 Manuscript organization

This habilitation manuscript is organized as follows, following the three main research axes of personal works. Chapter 2 presents contributions on computer-aided design techniques based on dataflow MoCs. Chapter 3 focuses on how the resilience of embedded systems to approximations can be characterized and exploited to build more efficient signal processing and AI systems. Chapter 4 details work on TPG for building low-power reinforcement learning agents. Finally, Chapter 5 concludes this manuscript with potential directions and challenges for future work.

Pppmppfpmmppppff
mfmppfppppppmmm mfmmfffpmmpp
ffmppffmf fmfpfm Pppmppfpmmppppff
mfmppfppppppmmm pmfmppfmp
ffmppffmf mpmppffppppp

Kenny McCormick, South Park

Chapter 2

Embedded Parallel Software Synthesis with Dataflow Models of Computations

Programming embedded Multiprocessor Systems-on-Chips (MPSoCs) is a tedious task that requires overcoming many challenges. The first challenge the developer often faces is how to efficiently and seamlessly capture the parallelism of the applications. To address this issue, there is a wide variety of parallel Models of Computation (MoCs) [88], each defining a mathematically grounded semantics suitable for specifying, verifying, analyzing, and optimizing a particular type of system [T5, 85].

In this chapter, we focus on the use of dataflow MoCs to specify parallel software for embedded MPSoCs. The chapter is organized as follows: Section 2.1 introduces the basic semantics of dataflow MoCs and presents the design flow used to build a system with them. Sections 2.2, 2.3, and 2.4 present the scientific contributions proposed for dataflow MoCs. Section 2.2 presents contributions on the semantics of dataflow models, extending their expressiveness and analyzability. Section 2.3 focuses on front-end analysis and optimization techniques, and Section 2.4 introduces novel DSE techniques based on dataflow semantics. Potential directions for future works on dataflow MoCs are outlined in Section 2.5 and detailed in Chapter 5.

This chapter summarizes and synthesizes content from the following personal publications: [J3, J9, B2, B3, C14, C15, C19, C28–C31, C39, C43]

2.1 Dataflow-Based Programming in Modern Computing Systems

A common semantics of all dataflow MoCs is the specification of systems with directed graphs called dataflow graphs or process networks [43]. Some elements found in this general dataflow semantics are

- **Actors**: The vertices of a dataflow graph, commonly called actors or processes, represent the computational entities of the dataflow MoCs. Each actor in the graph can consume and produce data on a set of data ports, and perform some processing using or generating that data.

- **Communication channels**: The edges of a dataflow graph represent communication channels used to transfer data between actors, usually assuming a First-in, First-Out queue (Fifo) mechanism.

- **Data tokens**: The atomic pieces of data that pass along the edges of a dataflow graph are called data tokens. The type of data abstracted by an individual data token depends on the specified system requirements. A data token can be a simple 8-bit integer, a complex number with two floating-point values, or even an entire 2D image with millions of pixels. Heterogeneous types of data tokens can coexist in a single dataflow graph, with each communication channel typically associated with a single type of data token.

Specifying the internal behavior of actors is not always an integral part of MoC semantics. Instead, dataflow MoCs typically specify a set of rules that govern when actors are allowed to consume and produce a certain number of tokens [59].

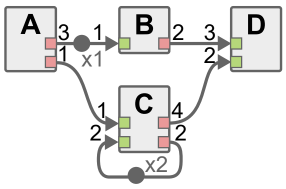
The popularity of dataflow MoCs for the design of stream processing systems stems largely from the advantages they offer for deriving efficient implementations on modern hardware and software technologies. Implementing efficient software on modern hardware requires allocating processing, memory, communication, and power resources to each part of the system. By clearly exposing separate computational entities, data movements, and computation triggers, dataflow MoCs facilitate this implementation process. Another key advantage of dataflow MoCs is their expressiveness for concurrent computation and data movement, which is an essential feature for both hardware and parallel software design [24].

The remainder of this section is organized as follows: Section 2.1.1 describes the semantics of dataflow MoCs that will be studied in later contributions, and Section 2.1.2 presents the typical design flow based on dataflow MoCs.

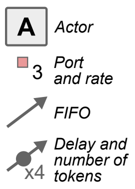
2.1.1 Dataflow MoCs

Synchronous Dataflow (SDF)

The SDF MoC was introduced by Lee and Messerschmidt [59] to model and optimize DSP applications on parallel hardware. The semantics and an example of SDF graph are presented in Figure 2.1.



(a) SDF graph example.



(b) SDF semantics

Figure 2.1: Synchronous Dataflow (SDF) example and semantics

Formally, an SDF graph $G = \langle A, F \rangle$ is a directed graph containing a set of actors A connected by a set of FIFOs F. prod(f) and cons(f) denote the actors in A that produce and consume tokens on a FIFO f, respectively. For each output FIFO $f \in F$ connected to an actor $a \in A$, a data rate is specified by the function $\text{rate}_{\text{prod}} : A \times F \to \mathbb{N}^*$. Symmetrically, $\text{rate}_{\text{cons}} : A \times F \to \mathbb{N}^*$ defines the consumption rate of an actor $a \in A$ on an input FIFO $f \in F$. These production and consumption rates specify the exact number of data tokens exchanged by the actor each time it fires and completes an atomic execution. As shown in 2.1b, production and consumption rates are denoted by an integer value written next to the data ports.

To ensure the liveness of cyclic data paths, that is, the absence of deadlocks when executing the SDF graph, delays must be used to specify the presence of data tokens in a FIFO at initialization time. The number of delays on each FIFO is specified by $\text{delay} : F \to \mathbb{N}$. For example, in the SDF graph of Figure 2.1a, the delay in the FIFO (C, C) ensures the liveness of the self-loop on the actor C. When executing a live SDF graph, an iteration of the graph occurs when all its actors have been executed at least once, and the number of tokens in each FIFO is back to the initial number specified by delays.

Dataflow forms of parallelism

The SDF execution is data-driven, meaning that an actor can fire as soon as it has enough tokens, as specified by its consumption rates on the input FIFOs. This semantics makes it easier to specify task parallelism, since two actors with enough tokens in their input FIFOs can be fired concurrently. For example, in the SDF graph in Figure 2.1a, each firing of actor A produces data tokens that can trigger executions of actors B and C, possibly in parallel. It may happen that the number of tokens available on a FIFO is sufficient to trigger multiple firings of its consumer actor. Since SDF actors do not maintain an internal state, there is no hidden data dependency between successive firings of an actor. Therefore, multiple firings of an actor can occur in parallel if enough input data tokens are available. An example of such data parallelism can be observed in Figure 2.1a, where actor A produces enough data tokens at each firing to trigger three firings of actor B. In addition to task and data parallelism, pipeline parallelism can also be explicitly specified using delays [59].

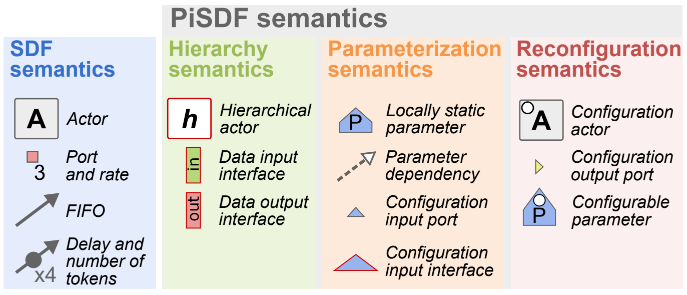
Static, dynamic and reconfigurable dataflow MoCs

SDF is a static MoC, meaning that all production and consumption rates are fixed at compile time. While static MoCs promote compile-time analyzability and optimizations, their restricted semantics, which is not Turing complete, hinders the specification of more dynamic applications. For this reason, *dynamic* and *reconfigurable* dataflow MoCs have been proposed. In both dynamic and reconfigurable MoCs, the data exchange rates of actors can change during the execution of an application. In dynamic MoCs, data exchange rates can change at any time during system execution, while in reconfigurable MoCs, reconfiguration can only occur at limited points in application execution [69]. This difference gives dynamic MoCs greater adaptability, but reconfigurable models have better predictability, making them more susceptible to compile-time and run-time optimizations.

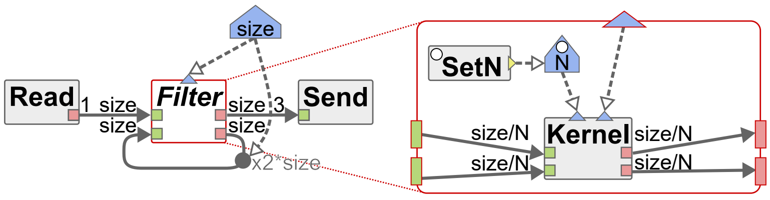
Parameterized and Interfaced SDF (PISDF)

PISDF is a reconfigurable dataflow MoC, whose semantics is presented in Figure 2.2. The PISDF semantics combine the semantics of SDF, a hierarchy mechanism [76], and an explicit parameterization tree with a reconfiguration mechanism [C22].

The hierarchy mechanism allows specifying the internal behavior of an actor with a dataflow subgraph instead of code for non-hierarchical actors. In the PISDF MoC, each data port of a hierarchical actor is viewed as a *data interface* to its subgraph. The purpose of an interface-based hierarchy [76] is to ensure the compositionality of the



(a) PISDF semantics



(b) PISDF graph example

Figure 2.2: Parameterized and Interfaced SDF (PISDF)

model, which makes a separate analysis of each subgraph equivalent to a global analysis of the entire hierarchy. Using the divide and conquer approach, compositionality can be used to speed up analyses and optimizations of the graph [C14, C15, 20]. To enable this compositionality, data interfaces automatically duplicate and discard data tokens if, during a subgraph iteration, the number of tokens exchanged on FIFOs connected to interfaces is greater than the number of tokens produced on the corresponding data ports of the parent actor.

The parameterization semantics of the PISDF MoC consists of a set of parameters P and parameter dependencies, configuration input ports, and interfaces. A parameter $p \in P$ is a vertex of the PISDF graph associated with an integer valuation function $\mathrm{val} : P \to \mathbb{N}$. The value associated with a parameter is propagated through explicit dependencies to other parameters and to actors that may use this value in expressions specifying their own values or rates of their dataflow ports. In the PISDF MoC, it is possible to disable all firings of an actor by setting all its rates to zero.

The reconfiguration semantics of PISDF MoC is based on special reconfiguration actors. When fired, reconfiguration actors dynamically change the value of a parameter in their graph. Reconfiguration actors must be fired exactly once per firing of their parent actor, before any non-configuration actor in their subgraph. This restriction is essential to ensure safe reconfiguration of the subgraph to which configuration actors belong. For example, in the PISDF graph of Figure 2.2b, when the hierarchical *Filter* actor is fired in the top-level graph, the *SetN* reconfigurable actor is fired first in its subgraph. The *SetN* actor sets a new value for the parameter *N*, which is used in the production and consumption rate expressions of the *Kernel* actor.

The short time between a reconfiguration and the execution of actors can be used to verify graph properties, such as the absence of deadlock, but also to make on-the-fly resource allocation decisions. The SPIDER runtime was developed [C27] to support the execution of PISDFs graphs on heterogeneous platforms.

2.1.2 Dataflow CAD flow

A simplified overview of a typical Computer Aided Design (CAD) flow based on dataflow MoCs is presented in Figure 2.3.

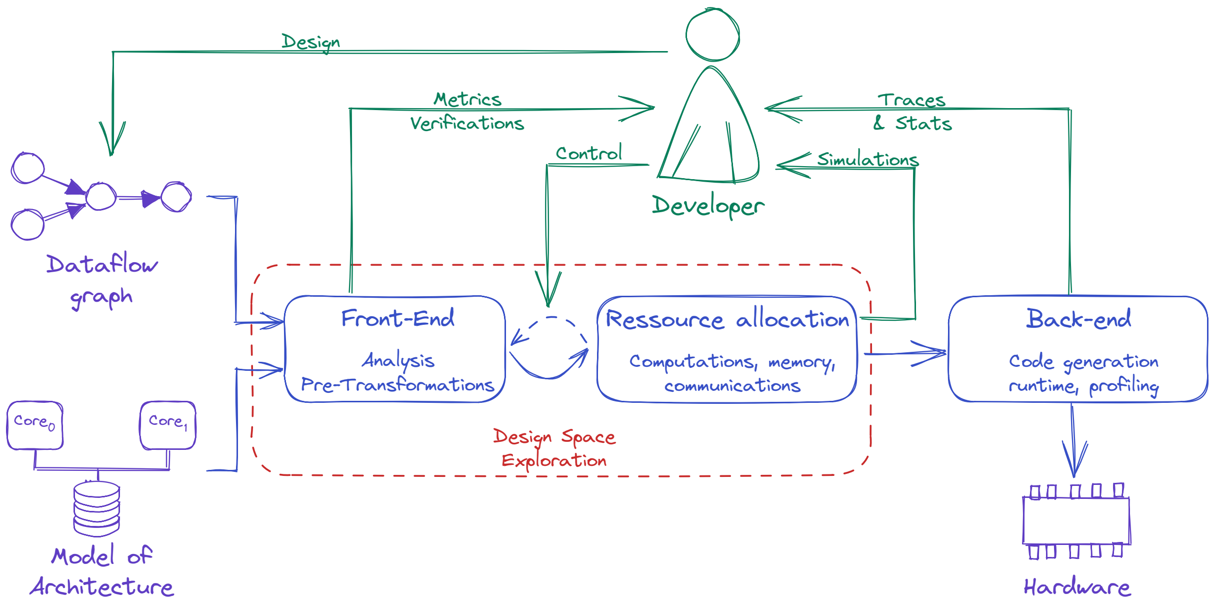


Figure 2.3: Typical dataflow-based CAD flow.

Inputs:

As illustrated in Figure 2.3, the CAD flow based on dataflow MoCs takes as input the dataflow graph of the application and a high-level Model of Architecture (MoA) of the target hardware [J12, C41]. These two input models are completely independent, which ensures portability of dataflow graphs to multiple hardware and usability of hardware for multiple applications. In addition to these inputs, a set of additional constraints is often required to capture the specific deployment constraints for a given pair of application and hardware.

Front-end:
The front-end part of the design flow performs analysis and model transformation based on the semantics of the input dataflow MoC. For example, a first analysis for static dataflow MoCs is to verify the consistency of the dataflow description [59]. A graph is said to be consistent if its infinite execution is possible without accumulating an infinite number of data tokens in any FIFO. Analyses can also be used to predict, early in the design flow, key properties of the design system, such as its latency [20], its throughput [C14, C15], or its memory footprint [C20].

Graph transformations applied during the front-end part of the design flow are often used to pre-optimize application graphs for resource allocation. For example, the single-rate transformation is often applied to dataflow models with well-defined production and consumption rates associated with actors, such as the SDF and PISDF models. The single-rate transformation produces a functionally equivalent application graph where all data parallelism is converted to task parallelism, and where each FIFO is written to and read from only once per execution of the dataflow graph. The allocation of processing and memory resources is much simpler for the single-rate graph than for the original graph. In fact, each actor is executed only once per execution of the single-rate graph, and each FIFOs has a fixed size and is either full or empty during graph execution.

Resource allocation:
Resource allocation consists of the temporal, logical, and spatial allocation of hardware resources that support the execution of the described dataflow application. When deploying a dataflow graph on an MPSoC, resource allocation includes: mapping and scheduling actor firings to processing elements, allocating FIFOs in memory, scheduling communications.

Together with front-end optimizations, resource allocation forms the Design Space Exploration (DSE) part of the dataflow MoCs design flow. As illustrated in Figure 2.3, supervised or unsupervised iteration of these two parts of the design flow is often used to gradually refine the deployment of a dataflow graph on a MPSoCs [C29, C43].

Back-end:
Once all design decisions have been made during the DSE process, they must be translated into an executable prototype by the back-end process. For static dataflow MoCs,

a code generation process is generally responsible for generating parallel code from the actor firing and communication schedule and memory allocation [C41, C43, B4].

For dynamic and reconfigurable dataflow MoCs, resource allocation is performed dynamically while the application is running. A lightweight runtime manager is therefore needed with dynamic MoCs to manage the DSE process and implement on-the-fly decisions [C27].

The following sections present personal contributions that address key challenges in all steps of the dataflow CAD process illustrated in Figure 2.3. Section 2.2 presents contributions that extend the expressiveness of dataflow MoCs. Section 2.3 introduces front-end analysis and transformation aimed at increasing the efficiency of the DSE process, itself studied and extended in Section 2.4.

2.2 Dataflow Model Extensions

The basic semantics of dataflow MoCs introduced in Section 2.1 has been extended many times, improving its expressiveness, analyzability, user-friendliness, and conciseness. An excellent survey of the state of state-of-the-art dataflow MoCs can be found in [85]. This section presents an overview of personal contributions to dataflow model semantics in Section 2.2.1, and a focus on a selected contribution in Section 2.2.2.

2.2.1 Contributions overview

Numerical representation for directed acyclic graph.
Building the single-rate equivalent of an SDF graph is a way of explicitly exposing dependencies across all actor firings of the original SDF graph. By ignoring delays, the single-rate graph becomes a Directed Acyclic Graph (DAG), further simplifying its analysis. Thus, the single-rate DAG is an intermediate representation where information is already pre-processed to help make simpler and faster scheduling algorithms. However, once the DAG is built, the scheduler no longer benefits from the compact and expressive representation of the original MoC used to describe the application. In fact, losing the compactness of the original graph can become problematic when building the DAG of graphs with a high degree of parallelism. In such massively parallel applications, the complexity of the DAG grows exponentially with the data parallelism, and so does the complexity of the scheduling problem. Building the single-rate DAG of a SDF graph is therefore not well suited for embedded runtimes, where scheduling must be done on the fly.

To address this issue, a numerical representation of dependencies between actors for the SDF and PISDF MoCs is introduced in [J3]. With this numerical representation, all data and reconfiguration dependencies between actor firings are computed on-the-fly from the original dataflow graph instead of building a DAG. We showed that the numerical representation is better suited for fast resource allocation of the application than DAG-based methods due to the cost of building and storing DAG. Experiments on various computer vision and machine learning applications showed significant gains over DAG-based methods, with on average a 97% smaller memory footprint of the intermediate model and a 7.8x acceleration of the resource allocation process.

Real-time extension.
As a synchronous MoC, the SDF semantics has no notion of execution or transfer time for actors and data. The execution of actors, and the associated consumption and production of tokens, occurs instantaneously if enough data tokens are available. While this untimed property is essential to support the implementation agnosticism of SDF, it makes the model ill-suited for specifying real-time systems.

In order to model the timing constraints of a real-time system, an extension of the SDF semantics has been proposed in [C29]. In the proposed extended semantics, each actor is associated with a WCET (Worst-Case Execution Time), and some actors have periodic release times with implicit deadlines, meaning that they must complete a firing at regular time intervals. While actors without periodic deadlines are *aperiodic* by definition, their data dependency on periodic actors imposes constraints on their scheduling. We say that such a graph has partially periodic constraints. In addition to extending the SDF semantics, [C29] also formalizes a condition for quickly assessing the schedulability of a graph without computing a schedule; and an offline non-preemptive scheduling algorithm that satisfies the periodicity and precedence constraints. Experiments show that the proposed non-preemptive scheduler is fast, scalable, and efficient, on par with equivalent state-of-the-art real-time schedulers.

Passive-active flowgraphs.
In applications specified with a dataflow graph, *special* actors are often needed to perform data reorganization. These actors may be used, for example, to transpose a matrix, to broadcast data from a single producer to multiple consumers, or to interleave data from individual producers into a single stream. While these actors are essential to the specification of any streaming application, they do not represent actual computations, but are nonetheless modeled as a black-box actor in the dataflow graph.

In [J9], a separation between active and passive blocks in dataflow semantics is introduced. Active blocks are regular actors that perform computations during their firing, while passive blocks represent actors that reorganize data. When executing the passive-active flow graph formed by these blocks, passive blocks do not need to be executed. Instead, passive blocks alter the way preceding or succeeding actors read and write data, according to the pattern specified by the passive block. Through model-based optimization, passive-active flowgraphs can be optimized, especially by merging chains of passive blocks. Simulation results show that the use of passive-active flowgraphs enables significant improvements in both throughput and memory footprint compared to equivalent dataflow models.

Nested `for` loops modeling

One can model single for loops by SDF graphs, as long as the loop can be divided into sub-parts that access chunks of data of equal size. However, there is no general technique for modeling multiple nested for loops in the SDF MoC, especially when the boundaries of the inner loops vary.

```
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < func(i); ++j) {
    output[i][j] = comput(input[i][j]);
}}
```

Listing 2.1: Perfect nested `for` loops example

The contribution in [C31] is the modeling, through SDF graphs, of multiple perfectly nested loops with explicit parallelism and variable bounds in their inner loops. An example of such nested `for` loops is given in Listing 2.1. In loops with explicit parallelism, there is no data dependency between successive iterations. Boundary variability means that the number, start, and end points of inner loop iterations vary as a function of the outer loop iterator. Perfectly nested loops perform computations only in the innermost loop.

To model perfectly nested `for` loops with an SDF graph, the technique proposed in [C31] first identifies the entire iteration space by simulating an execution of the nested loops. Then, the iteration space is cut into chunks of equal size to regularize the iteration space. Finally, iterations over these chunks are represented by an SDF actor that consumes chunks one at a time, but possibly multiple times in parallel thanks to data parallelism. Experiments show that computations modeled in SDF using this technique achieve competitive performance against *OpenMP* [C31].

2.2.2 Persistent States and Initialization in Hierarchical Dataflow MoCs

Dataflow MoCs have proven efficient for modeling continuous streaming applications with coarse-grained data and task parallelism. However, modern streaming applications, such as adaptive filtering or machine learning, integrate parameters that persist and are updated across multiple graph iterations. While dataflow delays [59] can theoretically be used to model persistent data, their behavioral semantics is often limited and cannot be used to uniquely specify the persistence scope and initialization of data.

To enable unambiguous specification of persistent data in dataflow MoCs, the State-Aware Dataflow (SAD) metamodel is introduced in [C5]. The SAD metamodel can be used to extend the semantics of any dataflow MoC that implements a well-defined notion of graph iteration. SAD adds both explicit initialization of delays and hierarchical state awareness through the use of a customizable persistence scope of delays to the extended MoC.

State-Aware Dataflow (SAD) initialization semantics

In the proposed semantics illustrated in Figure 2.4, a delay d is defined by a tuple $d = \langle f, n, cin, cout \rangle$, where $f \in F$ is the FIFO to which the delay is associated, $n \in \mathbb{N}$ is the number of initial tokens, and $cin, cout \in A$ are two optional data connections. Data connections *cin* and *cout* are equivalent to data ports of dataflow actors that can be connected to FIFOs to consume and produce data tokens, respectively. The input data connection *cin* of the delay associates a *Setter* actor, which is responsible for initializing the data tokens of f. The output data connection *cout* of the delay associates a *Getter* actor that receives the last values held by the delay. The data flow rates of *cin* and *cout* are such that $rate(cin) = rate(cout) = n$. However, the production rate of the *Setter* actor and the consumption rate of the *Getter* actor need not be equal to the rates of *cin* and *cout*.

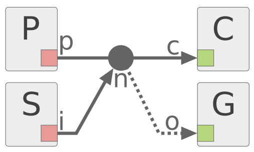


Figure 2.4: SAD: delay initialization semantics.

In Figure 2.4, actors S and G are the *Setter* and *Getter* actors of the delay associated with the FIFO between actors P and C, called *Producer* and *Consumer* of this FIFO. The new data connections introduced by SAD induce the following precedence rules in the firing order of actors during each graph iteration.

R1. All firings of the *Setter* actor of a delay must occur before the first firing of the *Consumption* actor of this delay.

R2. description All firings of the *Consumption* actor of a delay must occur after the last firing of the *Production* actor of this delay.

These rules can be easily enforced by the legacy dataflow resource allocation process using a graph transformation of the proposed delay semantics into the equivalent, but much more complex, SDF semantics [C5]. For example, Figure 2.5 shows the SDF graph that implements the same behavior as the one from Figure 2.4. The integer values $a, b \in \mathbb{N}^*$ represent the number of executions of actors P and C, respectively, during a graph iteration. This graph is valid under the condition that the number of delays n divides $p \times a = c \times b$, the number of tokens that are exchanged between actors P and C during a graph iteration. Several special actors must be introduced for this SDF graph to work: *Duplic.* actors that duplicate consumed tokens to satisfy production rates on their output ports, *Switch* actors that forward tokens received from S on its first execution and then those received from the feedback loop, *Iter*, which produces an increasing sequence of integer values needed to control the state-less *Switch* actor, *Last*, which forwards the last tokens it consumed to G, and *Swap*, whose internal datapath is depicted in the figure.

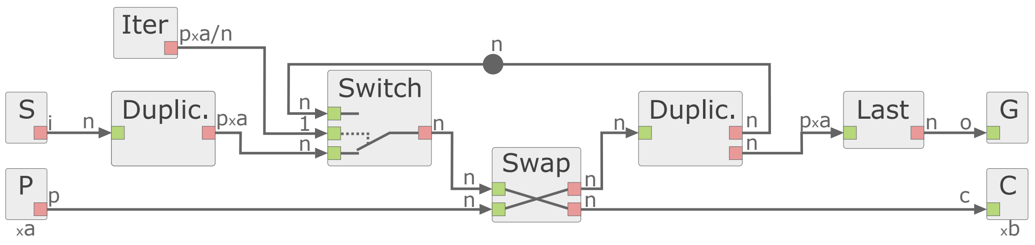
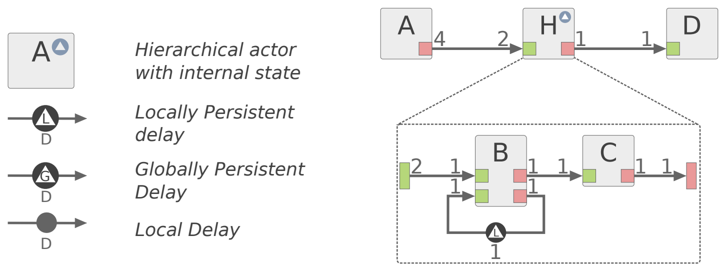


Figure 2.5: SDF equivalent to the SAD graph from Figure 2.4.

SAD persistence scope semantics

In hierarchical dataflow MoCs, such as the PISDF model presented in Section 2.1.1, the presence of delays in the hierarchy poses a persistence scope problem. Indeed, the presence of a delay in a subgraph G_H of a hierachical actor H induces an internal state. Such a state can either be discarded at the end of the firing of H or preserved for the next firing. To preserve the state of H, it is therefore necessary to extend the persistence scope of the delay outside the subgraph G_H with an unambiguous persistence semantics.


A
Hierarchical actor with internal state
L
D
Locally Persistent delay
G
D
Globally Persistent Delay
D
Local Delay
A
4
2
H
1
1
D
B
C
2
1
1
1
1
1
1
1
1
L
1

(a) Semantics (b) Graph example

Figure 2.6: SAD persistence scope.

The proposed SAD semantics [C5] for explicitly specifying the persistence scope of delays in hierarchical graphs is illustrated in Figure 2.6. There are three persistence scopes for delays in this semantics: local, locally persistent, and global. Local delays do not persist beyond one iteration of the graph to which they belong. Locally persistent delays persist for one level of the hierarchy, meaning that their value is propagated through successive firings of their parent hierarchical actor. Thus, locally persistent delays establish a precedence relationship for successive firings of the parent actor H of the subgraph G_H to which they belong. Globally persistent delays persist through all upper levels of the hierarchy. In Figure 2.6b, a locally persistent delay is used in the subgraph of H. As for the SAD initialization semantics, the contribution in [C5] provides rules for translating the different types of delays into legacy dataflow semantics.

SAD semantics benefits

The main benefit of the SAD semantics is that it allows unambiguous specification of persistent data initialization and scope in dataflow semantics. Although equivalent behavior can usually be described using legacy dataflow semantics, this requires expensive, complex, and non-generic dataflow patterns. For example, as shown in Figure 2.5, 6 special actors and 6 FIFOs are introduced in the SDF graph to mimic the effect of a single initialized delay. Contrary to the SAD semantics, the SDF pattern presented here is not generic, as it is not valid if, for example, n is neither a divisor nor a multiple of $p \times a$. Similarly, in [C5], a use-case analysis on a state-of-the-art reinforcement learning application shows a 35% reduction in allocated memory with SAD semantics compared to legacy PISDF.

2.3 Model-based front-end optimizations and analysis

A key advantage of dataflow MoCs is that the mathematically grounded semantics of models opens the way for powerful model-based analysis and optimization, early in the CAD flow. This section presents an overview of personal contributions to dataflow analysis in Section 2.3.1, and an emphasis on a pre-optimization technique for controlling the granularity of applications in Section 2.3.2.

2.3.1 Contributions overview

Throughput evaluation of hierarchical SDF graphs

During the design of stream processing applications, throughput is one of the key metrics to be evaluated and optimized as early as possible by the designer. Especially in real-time systems, evaluation of this metric can be used to identify timing problems early in the design flow, without going through the whole time-consuming DSE process. Very fast evaluation of application throughput is thus useful for providing real-time feedback to the developer during application development, or as part of the DSE process [86].

State-of-the-art throughput evaluation techniques for SDF graphs do not consider or exploit the hierarchy in the graph, as enabled by the PISDF semantics. Therefore, when evaluating the throughput of a hierarchical graph using these techniques, the hierarchy must first be flattened. Flattening a graph consists of replacing hierarchical actors with their subgraphs, potentially resulting in an exponentially large graph that is very complex to analyze.

Contributions in [C14] propose to exploit the hierarchy and the compositionality of the PISDF dataflow MoC to speed-up the computation of it theoretical throughput. The basic idea behind these contributions is that a bottom-up approach can be used, where levels of hierarchy are analyzed separately. To do so, the throughput of each lower-level sub-graph is analyzed. Then, instead of flattening the graph hierarchy, each hierarchical actor is replaced with a simple model, capturing the throughput characteristics of its sub-graph. Using this technique, the evaluation of the global throughput of a hierarchical graph is considerably sped-up, with minor or no loss of accuracy [C14, C15].

Automated Pipelining of SDF

The SDF semantics provides intuitive constructs for expressing task and data parallelism. While the SDF semantics also supports pipelining of applications through the use of delays, this type of parallelism is less intuitive to use and is often viewed by developers as a by-product of resource allocation rather than a developer's responsibility. Moreover, adding delays to a SDF graph requires extra care, as delays can alter and even corrupt the proper functioning of an application [77].

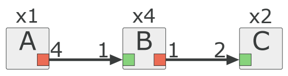
A fast heuristic algorithm for pipelining SDF graphs is proposed in [C30]. The heuristic automatically adds pipeline stages in the SDF graph in the form of delays, given the execution time of the actors and the number of Processing Elements (PEs). The heuristic is parametric: the developer can choose the number of pipeline stages to add. To select the best position to introduce a new pipeline stage, the proposed approach relies on simple as-soon-as-possible and as-late-as-possible topological orderings of dataflow graphs. Combined with the execution times of the actors, these topological orders provide all the information needed to balance the computational load of the created pipeline stages. The efficiency of the approach is illustrated by optimizing a set of signal and image processing applications running on multiple PEs. On average, when adding a pipeline stage, our heuristic selects a stage that results in better throughput than 90% of all possible stage placements.

2.3.2 Actor Clustering for Controlled Task Granularity

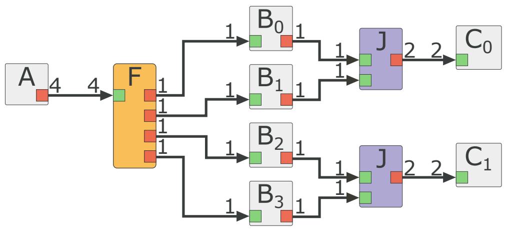
Many resource allocation methods for dataflow MoCs rely on complex graph transformations to expose their parallelism, which can result in complex graphs for embarrassingly parallel applications. For such applications, state-of-the-art mapping and scheduling techniques are prohibitively complex, while the exposed parallelism often exceeds the parallel processing capabilities of the target architecture. In this context, clustering heuristics can be used to preprocess dataflow graphs to reduce their complexity prior to resource allocation. Proposed in [C43], SCAPE is an automated clustering method that automatically adapts the granularity of a hierarchical PISDF graph to the parallel processing capabilities of the target MPSoC.

Clustering of SDF actors

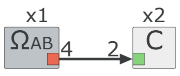
Clustering SDF actors consists of aggregating selected actors into groups that are considered as a single atomic actor during subsequent graph transformations and model-based resource allocation. In other words, clustering replaces the complexity of multiple actors firing and exchanging data tokens with different patterns with a simple "black box" processing unit: a cluster actor. The internal computations of a cluster correspond to calls to all clustered actors, which are typically managed by a single-processor scheduler, which is a polynomial-time algorithm [9].



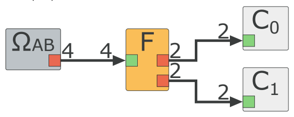
(a) Original SDF graph



(b) DAG expansion



(c) Clustered SDF graph



(d) Clustered DAG expansion

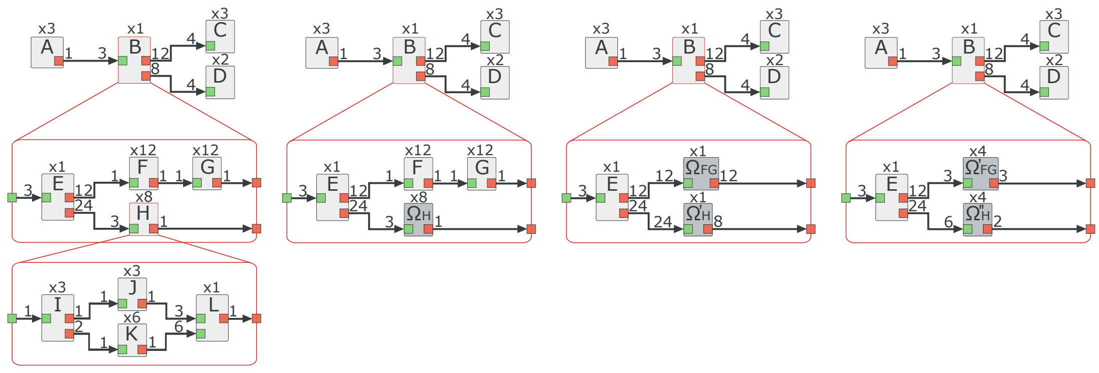
Figure 2.7: SDF clustering for controlled DAG complexity

A dummy example illustrating the benefits of SDF clustering is presented in Figure 2.7. The original SDF graph with three actors is presented in Figure 2.7a, and its DAG extension used for resource allocation in Figure 2.7b. Special actors F and J are introduced in the DAG to handle distribution and collection of data to multiple consumers and from multiple sources. In Figure 2.7c, a clustered SDF graph is presented where actors A and B have been merged into Ω_{AB}. The internal schedule of Ω_{AB} contains one call to A followed by four calls to B. The DAG obtained from the clustered SDF graph shown in Figure 2.7d is much simpler than the one obtained from the original SDF graph. While a simple DAG implies a faster allocation of resources, it is important to note that this may come at the cost of lost parallelism. For example, in Figure 2.7d, parallel executions of actor B is no longer possible. Unlike state-of-the-art clustering techniques that sacrifice application parallelism, the Scaling up of Clusters of Actors on Processing Element (SCAPE) method aims to preserve application parallelism so that it can exploit the parallel computing capabilities of the target architecture.

The SCAPE method

The SCAPE method is a parametric method where a parameter n_c is used to control the granularity of the clustered hierarchical SDF graph. The method consists of 3 steps, which are illustrated in Figure 2.8:

1. *Complete clustering of the* $(n_c - 1)^{th}$ *lower levels of the hierarchy.*
   Finer granularity of computations is simplified by clustering all subgraphs beyond a user-defined hierarchy level. The clustering is done using the acyclic pairwise grouping of adjacent nodes algorithm proposed in [9]. For example, fully clustering



(a) Input PISDF graph (b) Step 1 (c) Step 2 (d) Step 3

Figure 2.8: SCAPE method step-by-step example for a quad-core target.

the hierarchical PISDF graph of Figure 2.8a with $n_c = 2$ results in the graph in Figure 2.8b with 2 levels of parallelism remaining. Actors in the subgraph of the H actor have been clustered into the ΩH actor. When executed, ΩH sequentially executes the following actor calls *3I-3J-6K-1L*, where $3J$ means that actor J is executed three times.

2. *Partial clustering of the n_c^{th} level of the hierarchy.*
   The new bottom level of the hierarchy is partially clustered by identifying certain patterns of related dataflow actors that, when clustered, may still retain some degree of the data parallelism exploited in step 3 of the methodology. The first pattern identified is the Unique Repetition Count (URC) pattern, which is a chain of actors with the same number of executions per graph iteration. For example, in Figure 2.8b, the actors F and G form a URC chain, since both are executed 12 times during an iteration of their subgraph. The second pattern identified is the Single Repetition Vector (SRV) pattern, where a single actor is not part of an URC chain, and its number of executions is greater than the number of target processing elements. For example, in Figure 2.8b, actor H is an SRV actor. Both the URC and SRV patterns are replaced by a single cluster actor executed once, simplifying the equivalent DAG used for resource allocation. The graph resulting from the second step of the methodology is illustrated in Figure 2.8c. The actors F and G are clustered into the actor Ω_{FG}, whose unique firing corresponds to the sequential execution of *12F-12G*.

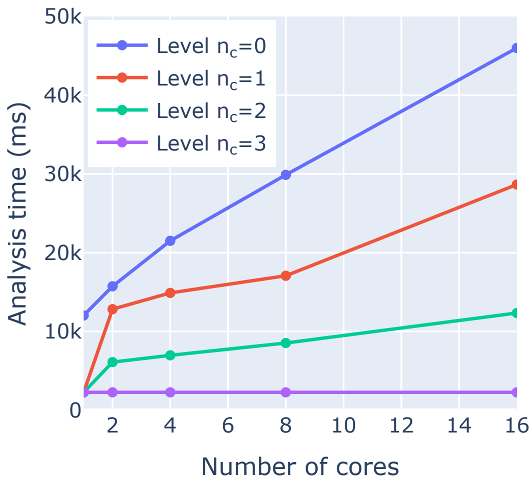
3. *Architecture-aware optimization of parallelism at the n_c^{th} level of the hierarchy.*
   The purpose of this last step is to change the number of firings of actors clustered

in step 2 to match their data parallelism to the parallel processing capabilities of the target platform. To do this, some of the data parallelism that was hidden inside the cluster during step 1 is brought back to the clustered actors. This is accomplished by changing the number of iterations of the clustered actors to match the number of processing elements in the architecture. For example, assuming a target architecture with 4 cores, the Ω'_{FG} actor is modified to be executed 4 times, where each firing corresponds to the sequential execution of *3F-3G*.

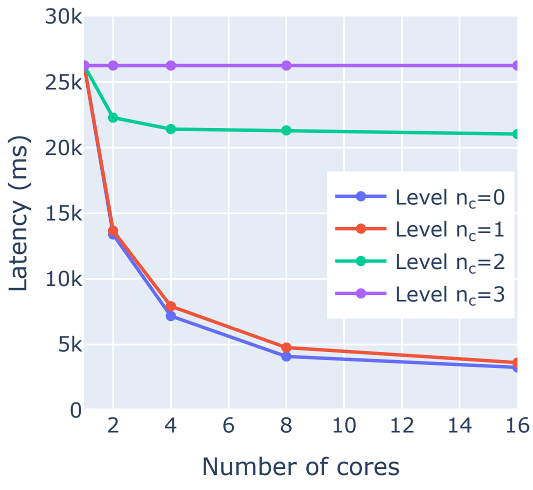
Using this technique, the complexity of the equivalent DAG is effectively controlled from 137 vertices for the input graph of Figure 2.8a to 41 after step 1, 11 after step 2, and 17 after step 3.

DSE time vs latency trade-off

Experimental results with SCAPE demonstrate the trade-off between the complexity of the resource allocation process and the quality of the produced results. Figure 2.9 presents experimental results for a stereo matching image processing algorithm described with a 2-level hierarchical PISDF graph [J5]. Experiments were performed with $n_c \in 0, 1, 2, 3$, and mapping and scheduling computations on homogeneous multi-core architectures with $nb_{core} \in 1, 2, 4, 8, 16$.



(a) Host Analysis Time



(b) Produced Latency on the Target

Figure 2.9: Analysis time and latency trade-off for SCAPE configurations.

Figure 2.9a shows the resource allocation time as a function of the number of cores targeted, for different levels of n_c with SCAPE. When $n_c = 0$, no clustering is applied, which leads to larger DAGs and longer analysis time. On the contrary, the larger the value of n_c, the more clustered actors there are and the shorter the analysis time.

Figure 2.9b plots the simulated latency of generated multi-core schedules as a function of the number of cores targeted, for different values of n_c with SCAPE. At lower values of n_c, most of the fine-grained parallelism of the application is preserved, resulting in better, that is, shorter, latency of the produced schedules. An extreme case can be observed with $n_c = 3$, where all actors of the graph are clustered into a single atomic actor, whose firing is equivalent to a single-core execution. Finally, an interesting trade-off is obtained with $n_c = 1$, where the analysis time is almost halved, as shown in Figure 2.9a, but the schedule latencies produced remain equivalent to those of the unclustered application graph. Further results on another application with a different profile are published in [C43].

2.4 Model-based Design Space Exploration (DSE)

To execute an application modeled with a dataflow MoC, hardware resources must be allocated spatially, logically, and temporally. Dataflow MoCs are well suited for this process, because computations are already separated into well-identified actors and memory requirements are modeled as FIFO buffers. Nevertheless, each of these resource allocation problems is NP-hard. Therefore, intelligent DSE strategies must be developed to tackle these problems. Section 2.4.1 presents an overview of personal contributions on dataflow-based DSE algorithms, and Section 2.4.2 details a contribution on the use of moldable parameters for multi-objective DSE optimization.

2.4.1 Contributions overview

Static distributed memory allocation

Optimizing the allocation of memory to support the execution of an SDF graph was one of the main contributions of my doctoral thesis [B4]. The methodology used to allocate memory for a static dataflow graph is based on a graph called Memory Exclusion Graph (MEG) that models the memory reuse possibilities. Each vertex of the MEG represents a buffer of a DAG, and the edges of the MEG model the impossibility of allocating two buffers in overlapping address spaces, mostly because the lifetimes of the data they store may overlap. Based on the MEG, bounding algorithms [C20] and shared memory allocation algorithms [J5, C21] were proposed during my thesis.

As a continuation of this work, an extension of the memory allocation process to support distributed memory has been proposed [C19]. To support distributed memory, a MEG is built for each distributed memory bank of the target architecture, based on the mapping and scheduling of dataflow actors. Experiments show that even in the presence of shared memory and private caches, the use of distributed memory closer to the cores significantly accelerates computations, up to 380% without caches and up to 19% with caches, on embedded MPSoCs [C19].

Reconfigurable dataflow runtime manager for many-core architectures
The PISDF model proposed in my PhD thesis is a reconfigurable dataflow model whose behavioral semantics allows runtime changes in data paths and computational workloads [B3, T4]. To support these dynamic changes, a runtime manager is required to keep track of parameter values in the executed graph and make on-the-fly graph transformations and resource allocation decisions. For the PISDF graph, the Spider (Synchronous Parameterized and Interfaced Dataflow Embedded Runtime) runtime manager has been proposed by Heulot et al. in [C27].

Based on Spider, the first embedded runtime manager allowing the execution of reconfigurable dataflow graphs on a many-core NUMA (Non-Uniform Memory Access) architecture was proposed in [C39]. The proposed runtime is based on new scheduling, synchronization, and memory allocation algorithms specifically designed for clustered architectures, where PE are grouped into clusters, each with a private memory bank. Experiments on *Kalray*'s 256-core *MPPA* architecture demonstrate the viability of the proposed runtime and its great potential, with energy efficiency up to 10 times better than a desktop CPU.

2.4.2 Multi-criteria Optimization of Algorithmic Design Choices with Moldable Parameters

When developing an application, developers often want to experiment with different configurations. In practice, different configurations can differ in many ways, from functional differences of alternative algorithmic choices to non-functional differences that only affect, for example, the degree of parallelism of a specification. This work aims to facilitate and automate the exploration of multiple configurations for dataflow applications.

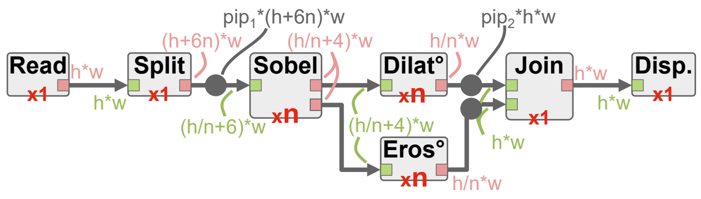
Most related works on DSE for applications modeled with static dataflow graphs assume that the dataflow graph is fixed before entering the DSE process. Thus, for a given application graph, the DSE process is responsible for evaluating multiple solutions provided by the resource allocation solvers. To explore different configurations with such a DSE process, the developer must manually modify the application graph, possibly by changing its static parameters, and restart the entire DSE process for each configuration. Few works consider exploring design choices on the application model itself, exploiting the dataflow MoC semantics. MASES [89] is one of them; it optimizes the throughput, latency, and processor utilization of applications represented with a restriction of the SDF MoC, where it automatically adds software pipelining.

This contribution [C28] goes beyond the state of the art by introducing new *moldable* parameters in the MoC semantics to explicitly specify different configuration alternatives. Building on these parameters, the multi-criteria DSE process can be automated to automatically find the set of Pareto-efficient configurations.

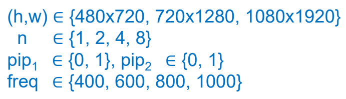
Moldable parameters

Parameters in the PISDF MoC can be used to set various characteristics of the application: data production and consumption rates on FIFOs, delay sizes, execution times and energy per actor firing, and even actor static integer input arguments. Tuning such single expression parameters is cumbersome for developers, as they must manually set the correct expression of a parameter to perform application analysis or code generation for each application configuration.

Moldable parameters are a simple extension of parameters as defined in the PISDF semantics [C22]. Each moldable parameter contains a list of symbolic expressions separated by semicolons. The first expression is the default, so moldable parameters can always be used as regular parameters. Just like regular parameters, symbolic expressions held by moldable parameters can be a simple static integer value, or a complex expression that depends on other parameters using mathematical operators and functions. A *parameter configuration* of the application dataflow graph is obtained by selecting and evaluating for each moldable parameter a single expression from the list of available ones.



(a) Parameterized graph



$(h,w) \in \{480x720, 720x1280, 1080x1920\}$
$n \in \{1, 2, 4, 8\}$
$pip_1 \in \{0, 1\}$, $pip_2 \in \{0, 1\}$
$freq \in \{400, 600, 800, 1000\}$

(b) Moldable parameter values

Figure 2.10: Dataflow graph with moldable parameters.

An image processing dataflow application that embeds moldable parameters is given in Figure 2.10. The behavior of the graph in Figure 2.10a is configured using 6 moldable parameters: h and w which control the processed image resolution, n which controls the number of image slices processed in parallel, pip_1 and pip_2 which can be used to activate different pipeline stages, and *freq* which represents the frequency of the target processor. Only the h and w parameters affect the output produced by the described application. All other parameters are called non-functional parameters, because they change the way the application can be executed, but not the result it produces. The list of values for each moldable parameter is given in Figure 2.10b. Considering all possible combinations of moldable parameter values, there are 192 configurations for this simple application example.

Multi-criteria exhaustive DSE

Running an exhaustive DSE for an application graph with moldable parameters consists of performing resource allocation for each configuration. The result is a set of metrics for each configuration that can be used to evaluate the efficiency of each configuration. Common optimization metrics include latency, throughput, power, and memory usage.

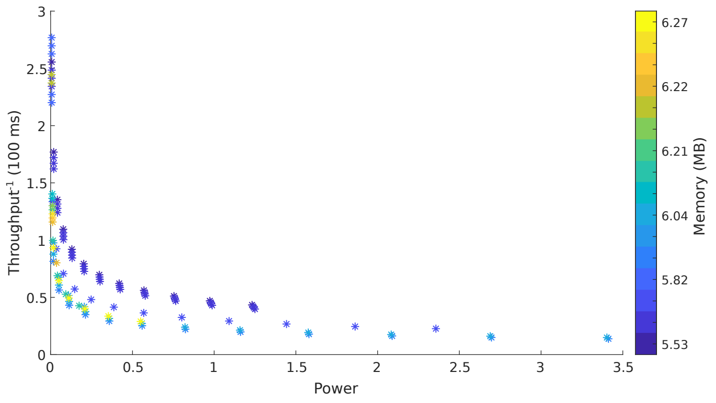


Figure 2.11: Pareto-front (Power, Throughput^{-1}, Memory) of SIFT for 2 pipeline stages.

Figure 2.11 presents an excerpt of results for a SIFT computer vision algorithm modeled with a moldable dataflow graph [C28, C31]. The plot represents the value of 3 optimization criteria to be minimized: Power, Throughput^{-1}, Memory footprint, for a fixed pipelining configuration of the application.

Among all the solutions generated by the exhaustive DSE process, only Pareto-efficient solutions are presented in Figure 2.11, that is, solutions that cannot be improved on one criterion without sacrificing another. These Pareto-efficient solutions are the ones sought by application developers, as they represent the best possible tradeoffs between observed metrics. As shown in [C28], a limited subset of moldable configurations leads to such Pareto-efficient solutions, with only 1% of the 10k configurations for the large SIFT application, and up to 20% of the configurations for smaller applications such as the one shown in Figure 2.10.

Towards smarter DSE
Exhaustive exploration of all moldable configurations is impractical. The heuristic algorithms responsible for mapping and scheduling dataflow computations and allocating memory take seconds to minutes to run for each configuration. In cases where there are thousands of configurations, exhaustive exploration can take hours or days, which is highly impractical.

Since only a few moldable configurations are Pareto efficient, a smarter DSE algorithm should seek these configurations without relying on exhaustive exploration. While such a DSE algorithm has not yet been built, a preliminary study [C28] shows that for many moldable parameters, it is possible to identify how variations in the parameter value affect the observed optimization criteria after resource allocation. We classify moldable parameters into 4 categories according to their influence on the optimization criteria:

- **Same** $\rightarrow$: The criterion is constant when the parameter changes.
- **Increase** $\nearrow$: The criterion increases strictly as the parameter value increases.
- **Decrease** $\searrow$: The criterion decreases strictly as the parameter value increases.
- **Inconsistent** $\nearrow\!\!\!\!\!\searrow$: The variation of the criterion is not strictly monotone, while the variation of the parameter value is monotone.

In [C28], it is empirically shown that a reliable classification of the influence of moldable parameters on optimization metrics can be built by exploring less than 15% of the DSE space. A goal for future work is then to build on such a classification to drive the search for moldable configurations that lead to Pareto efficient solutions.

2.5 Summary

In this chapter, contributions to design automation techniques for embedded software have been reviewed. These contributions are based on several dataflow MoCs whose semantics can be used to capture characteristics of parallel stream processing applications.

Building on this semantics, powerful optimization techniques are proposed to automatically build highly optimized system prototypes that rival and often surpass handcrafted software.

The proposed contributions fall into three main categories:

- Model extensions that extend the semantics of the dataflow MoCs to improve its expressiveness and analyzability.
- Front-end optimizations and analysis that exploit dataflow semantics early in the design flow to provide rapid feedback to the user or to simplify and accelerate the subsequent design flow.
- Model-based DSE techniques responsible for transforming high-level dataflow graphs into multi-optimized prototypes on modern MPSoCs.

The contributions presented here have been made possible thanks to the hard work and dedication of the following individuals:

- **PhD students**: Florian ARRESTIER, Hamza DEROUI, Julien HASCOËT, Alexandre HONORAT, Ophélie RENAUD.
- **Master Students**: Hugo MIOMANDRE, Thomas BOURGOIN, Dylan GAGEOT.
- **Post-Doc and engineers**: Clément GUY, Julien HEULOT, Antoine MORVAN.

Research perspectives on dataflow MoCs include work on distributed multi-node HPC and swarm systems, on dynamic neural networks, and on multiobjective optimizations. These directions are described in detail in Chapter 5.

I am Groot.

Groot, Planet X

Chapter 3

Resilience of Signal Processing Systems to Approximations

Integrated circuits packed with transistors implementing logic circuits using binary signals are the foundation of modern computing systems. With such hardware, computations must be modeled with finite-length binary words, which cannot support the infinitesimal precision of mathematical models. For example, with the 32-bit IEEE754 `float` format, the closest value immediately above 1 that can be represented is 1.00000012. Any operation that results in a number between 1 and 1.00000012 is quantized, that is, either thresholded or rounded to one of these values. While such approximations can be neglected most of the time, they can also be the cause of catastrophic system failure, such as the instability of a closed-loop system. When implementing a signal and image processing system, a key task of the designer is to translate the mathematical equations governing the operation of the system into digital computations, which may require the introduction and control of approximation errors. These approximations may concern the data formats, but also the arithmetic operations or even the implemented algorithm.

This chapter presents a series of contributions aimed at characterizing, optimizing, and exploiting approximations made in the design of complex signal processing systems. The chapter is structured as follows Section 3.1 briefly introduces the necessity, risks, and benefits of approximations in modern signal processing systems. Section 3.2 focuses on work done to accelerate the characterization of errors caused by approximations in simple and complex signal processing systems. In Section 3.3, we study how the resilience of AI to degraded data can be exploited to reduce the bitrate of images transmitted in the context of Video Coding for Machines (VCM). Finally, Section 3.4 concludes this chapter.

This chapter summarizes contributions and gathers content from the following personal publications: [C6, C9, C37, C38, P1, T2].

3.1 Introduction to Approximate Computing

3.1.1 Why is it Often Preferable to Make (small) Errors?

The most common approach to designing a computing system is to design the most accurate system possible. For example, when prototyping a signal processing algorithm with Matlab, NumPy, or Julia, the default data type used for real numbers is the `double` type. In fact, it can unerringly be assumed that miscalculations made with this 64-bit format are negligible in most algorithms. While the choice of a high-accuracy data format is often made unthinkingly, the deliberate pursuit of higher accuracy is also very common. A striking example of this pursuit is the ongoing competition to build the AI model that achieves the best possible accuracy in computer vision tasks, such as image classification on public datasets [52, 53].

Contrary to common practice, striving for higher accuracy is often unnecessary and may even be counterproductive or harmful in some applications. An antique example of a useful approximation is Archimede's approximation of $\pi \approx 22/7$. Since antiquity, this approximation of π has been used extensively and without harm in many fields, including construction and architecture, because of its simplicity [98]. For more modern and sensitive computations, higher accuracy may be required. For example, NASA uses 15 decimals of π, which is sufficient for their highest accuracy calculations for interplanetary navigation [25][1]. Another famous example of a useful approximation is the inverse square root calculation used in the 3D graphics engine of *Quake 3*. This fast algorithm for computing $1/\sqrt{x}$ is based on a hack of the 32-bit binary `float` numbers [63], followed by an iteration of Newton's algorithm. The algorithm produces an approximation of the result with a maximum relative error of 0.00175228, which is perfectly acceptable in most cases.

Useful approximations can also be made at the algorithmic and application level. An example of such an approximation is proposed by Muvva et al. in [68], where they evaluate several machine learning-based algorithms for controlling a drone that visually tracks and follows another drone. In this example, the authors show that it is sometimes preferable to have a less accurate but fast tracking algorithm than a more accurate but slightly slower one, with the latter losing its target more often than the former.

[1]This coincidentally corresponds to the accuracy of a `double` number for π.

While approximations are often used to get results faster, as shown in the previous examples, another benefit is often found on the energy consumption side. In fact, at the computational level, high-precision results require more resources to produce than approximate results. An interesting proof of this assertion is the difference in the energy used to compute `int`, `float`, and `double` multiplications on a standard CPU. As shown in [101] for a big.LITTLE *Arm* architecture with 4 A7 cores and 4 A15 cores, the energy used for a `float` multiplication is between 9% and 102% higher than for a 32-bit `int`, and for a `double` multiplication it is between 23% and 200% higher, despite the presence of floating-point units in all cores. In addition, `float` and `double` multiplication instructions have a latency between 33% and 133% longer than their `int` counterpart. Therefore, in such an architecture, using fixed-point arithmetic, where real values are encoded with `int` data types, can result in both a lower power footprint and higher performance at the expense of computational accuracy.

In summary, approximations can be used in system design to build more cost-effective systems where accuracy and quality of service are kept to an acceptable minimum.

3.1.2 Approximation Opportunities and Trade-offs

Many approximation opportunities have been studied in the scientific literature for the design of computing systems. As illustrated in fig. 3.1, these approximations can generally be categorized along three axes [11]: data, hardware, and algorithmic approximations.

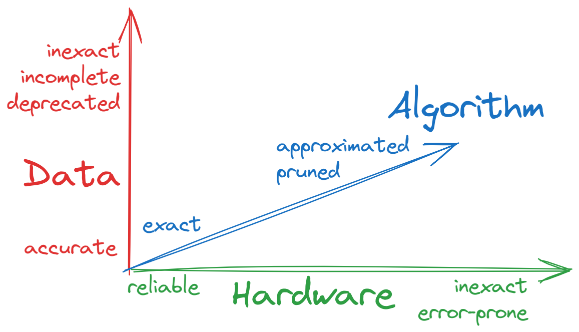


Figure 3.1: Approximate computing optimization axes [11].

Data approximations take many forms in the literature, with approximation levers encompassing data quality, quantity, and temporality. Lowering data quality consists of adopting storage formats that lose some accuracy of the data. As discussed in Section 3.1.1, a controlled loss of accuracy for real numbers can be achieved by adopting standard data types with narrower bit widths, such as 32-bit `float`, half-precision 16-bit `float16` formats [36], or even floating-point formats with custom widths [65]. Another alternative is to convert real numbers to fixed point formats of variable width, which can be manipulated using integer ALUs (Arithmetic and Logic Units) [60, 91]. Reducing the amount and temporality of data consists of neglecting some available input data of an algorithm, which reduces the number of energy- and time-consuming data accesses and synchronizations within a computing system. For example, a common way to reduce the amount of data in image processing algorithms is to reduce the resolution of the processed images by subsampling the input images [C38, T2, 47, 53]. Working with outdated data rather than fetching the most current data is another strategy for reducing the number of data accesses a computer system makes [82].

Hardware approximations take two main forms: inexact logic circuitry and error-prone working conditions. An inexact logic circuit implements a function with hard-wired shortcuts compared to the exact version. Because of these shortcuts, the circuit deterministically produces computational errors for some input values, but uses less silicon area, consumes less power, and is faster than its exact counterpart [C6, C11, 40]. Using hardware in error-prone conditions consists of using an exact circuit with unsafe voltage and frequency parameters. For example, using a circuit slightly below its minimum nominal voltage [80] allows significant power savings while introducing statistically harmless computational errors. A complementary approach, timing speculation, consists of increasing the clock frequency of a circuit beyond its safe limit [41]. Timing speculation can be used with [3] or without [41] error correction mechanisms, and allows further reduction of power consumption when combined with undervoltage powering of the circuit.

Algorithm approximations are modifications to the algorithms executed by a system that deliberately reduce the accuracy provided in exchange for reduced complexity. Algorithm approximations can be introduced by modifying the control flow of an application, for example by skipping iterations of a `for` loop [C8], or by choosing an early exit from an iterative algorithm [C8, 63]. At a higher level of abstraction, removing computations that do not dramatically affect the result of an algorithm [70]. In recent years, the popularity of this technique has increased dramatically with the pruning of Deep Neural Network (DNN) [60]. Another algorithm approximation technique is simply to replace all or part of an algorithm with an alternative algorithmic solution that provides a different tradeoff between the delivered Quality of Service (QoS) and the optimized non-functional properties that are being optimized.

Each of the above approximation techniques introduces potential or systemic inaccuracies and errors into the system in exchange for a non-functional benefit. The most commonly considered benefits are energy savings, reduced execution time or increased throughput, reduced silicon area, and reduced memory footprint of the system. Although introduced errors may be completely benign to the system, they most often degrade the QoS of the system, and may even be the cause of catastrophic failures, especially in closed-loop systems where even small errors can propagate rapidly [51].

3.1.3 Research Challenges for Approximate Computing

At a time when frugality is paramount, approximate computation remains a very active research area along its three research axes. Most of the ongoing open challenges in this domain concern the following 4 questions:

- **How to harmlessly approximate subparts of a system?** Creating new approximation techniques remains a fundamental open challenge to provide system designers with new optimization possibilities.

- **How to expose approximation opportunities to developers.** Many efficient approximation techniques have been proposed in the scientific literature for decades, but are not used by system developers. The reason for this non-use is that high-level programming languages and development tools simply do not expose or recommend these approximation techniques in their features. Thus, the use of approximation techniques requires a voluntary approach by the developer, who must identify candidate regions of his code and manually write the low-level code required to implement the optimizations.

- **How to predict the impact of local approximations on the global system.** When implementing approximations within a system, their separate and combined effects on the proper functioning of the system must be predicted in order to assess their viability. While formal methods exist to mathematically model the effect of approximations within a chain of computations, these formal methods are often not applicable [C11] in large systems with complex and dynamic control and dataflows. Therefore, new formal, statistical, or empirical methods are needed to quickly predict the effects of approximations in complex systems.

- **How to automate the Design Space Exploration (DSE) for optimal approximations?** The design space to explore when selecting and implementing approximations within a system is immense. The DSE requires considering many design choices, various constraints to satisfy, and multiple criteria to optimize [C7, C28, 66]. In this context, DSE consists of choosing the right combination of approximations, each of which offers several degrees of approximation to choose from, and each of which affects system performance and QoS in its own way. Each tested configuration must be evaluated, which usually requires running complex analyses or simulations [C9, C11]. Therefore, intelligent heuristics are needed to find the Pareto optimal solutions with respect to various metrics, such as performance, energy, and QoS, with as few evaluations as possible, while respecting the design constraints.

The personal contributions that study the impact and use of approximations in different contexts are presented in the following sections. Section 3.2 introduces fast techniques for modeling the influence of approximations on the behavior of a system. Then Section 3.3 examines how image approximations, that is, images degraded by lossy compression, can be exploited in the context of Video Coding for Machines (VCM).

3.2 Fast Error Characterization for Approximate Computing Systems

When implementing approximations within a computing system, it is essential to characterize how these approximations affect the functioning of the system. Indeed, during the DSE process, quantifying the impact of approximations helps the designer select the appropriate degree of approximation to introduce in each part of the system. Providing this characterization quickly is important to allow faster DSE and evaluation of more alternative configurations.

A circuit-level approximation characterization technique is presented in Section 3.2.1, and a system-level approximation characterization technique is presented in Section 3.2.2.

3.2.1 Circuit-Level Error Characterization with Inferential Statistics

The goal of this contribution is to characterize the frequency and amplitude of errors made by hardware approximate operators using inferential statistics.

Motivations & background:
An approximate operator is a logic circuit that takes operands as inputs and produces a result with a predefined word length, such as a 32-bit adder. Compared to their exact

counterparts, approximate operators are modified at the gate level to gain speed, power, or silicon area [41, 62].

An illustration of the computational errors made by an approximate adder is given in Figure 3.2. The simple approximate circuit considered is an 8-bit LOA (Lower-part OR Adder) adder [64], where the added operands are divided into two parts, the lower 5-bit part and the upper 3-bit part. The sum of the lower part is approximated with a simple bitwise **OR** operator, while the higher part is summed with a full adder. The heatmap presented in Figure 3.2 depicts as color intensity the error made for each pair of operand values, relative to the exact results: $RelativeError = (Result_{exact} - Result_{approx.})/Result_{exact}$. This adder produces large relative errors, up to 50% of the exact result, for small operand values. Overall, this operator produces the exact result only 24% of the time, but its average absolute relative error with respect to the exact result is only 3.77%.

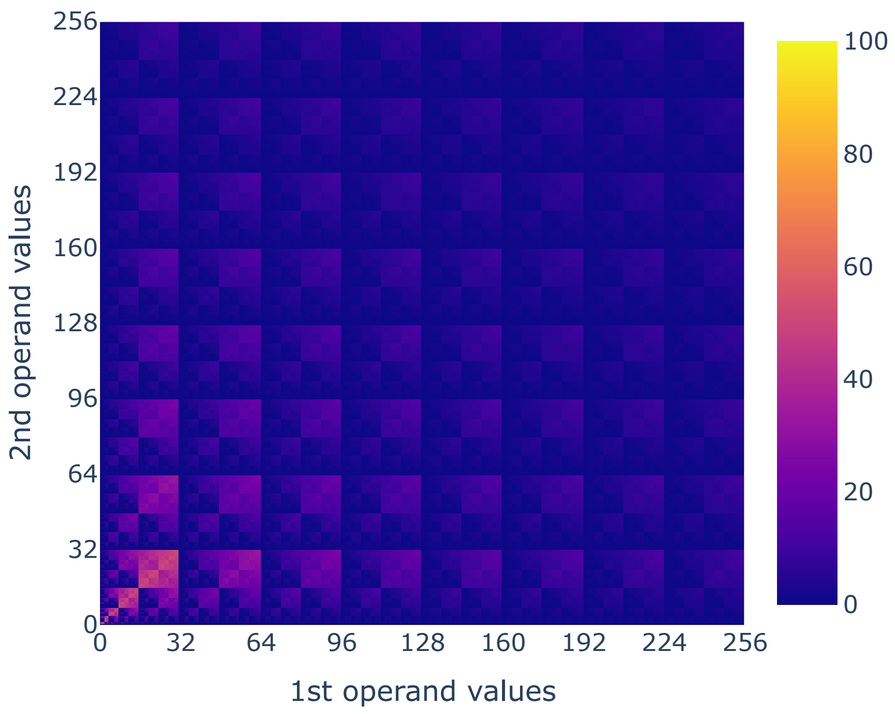


Figure 3.2: Relative error for an 8-bit LOA approximate adder [64]. Color intensity represents the error made for each pair of operand values, relatively to the exact results.

Characterizing the error introduced by an approximation circuit is not straightforward. While analytical methods exist to mathematically model approximation errors [61], these methods require time-consuming expert analysis of the circuits, and are often specific to a given operator with predefined circuit modifications. An alternative approach to characterize the error of these circuits is to measure the errors produced by the circuits by simulating their operation for different operand values. Unfortunately, bit-accurate hardware simulation is a computationally intensive process that can take between 300 and 4000 times longer to emulate than running a native addition or multiplication with the same CPU [C6]. However, since the simulation process can be automated, this method is often preferred over the analytical approach.

To characterize the frequency and amplitude of errors with simulations, a first option is to simulate the operator exhaustively, for all possible operand values, as done in Figure 3.2. For a 2-operand operator with a word length of n bits, the number of simulations to run is

$$Nb_{simulations} = (2^n)^2$$

While the number of input values to simulate remains reasonable for operators with short word lengths, for example 65 536 values for an 8-bit operator, this number becomes prohibitively large for larger ones, for example $\sim 4 \cdot 10^9$ values for a 16-bit operator, and $\sim 18 \cdot 10^{18}$ values for a 32-bit operator. Such an exhaustive simulation is simply not feasible for large operators. For example, assuming an optimistic simulation time of 1µs on one core, per pair of input values, and using 500 parallel cores, more than a year of simulation would be required to cover all possible values. Even taking into account symmetries to reduce the number of simulations to be run, which requires expertise in circuit analysis, exhaustive simulation remains prohibitively expensive.

Statistical methods are therefore needed to derive approximate but reliable error models from as few simulations of an operator as possible.

Characterization with Adaptive Sample-Size Inferential Statistics (Cassis)

The goal of the Cassis method [C6] is to automatically select the minimum number of random operand values to simulate in order to obtain a trustworthy error characterization of a circuit. The error model obtained for the analyzed circuit consists of a mean error distance (μ), which is the average absolute error made by the approximation circuit, and an error rate (f), which represents the probability that a result produced by the circuit is inaccurate. Formally, for a circuit with two n-bit operands, these metrics are expressed as follows:

$$\mu = \frac{1}{2^{2n}} \sum_{i=0}^{2^n-1} \sum_{j=0}^{2^n-1} |res_{exact}(i,j) - res_{approx}(i,j)|$$

$$f = \frac{1}{2^{2n}} \sum_{i=0}^{2^n-1} \sum_{j=0}^{2^n-1} err(i,j), \text{ where } err(i,j) = \begin{cases} 1 \text{ if } res_{exact}(i,j) \neq res_{approx}(i,j) \\ 0 \text{ otherwise} \end{cases}$$

where $res_{approx}(i,j)$ and $res_{exact}(i,j)$ represent the approximate and exact results, respectively, for the circuit under consideration. For example, for the LOA adder from Figure 3.2, the exact error characteristics obtained from exhaustive simulations are $\mu = 7.75$ and $f \approx 0.24$. While the previous equations assume a 2-operand circuit, a generalization to N operands would be straightforward.

The confidence criteria of the CASSIS method are set by the user, with the desired confidence constraints defined as an accuracy constraint h and a confidence constraint p, both expressed as probabilities. Thus, the method automatically returns a result $\mu_{estimated}$ such that there is a probability p that $|\mu_{estimated} - \mu_{exact}|/\mu_{exact} \leq h$.

CASSIS builds on inferential statistical techniques [87] to automatically derive the minimum number of simulated random input values to estimate the mean error distance and error rate within the confidence interval defined by $\langle h, p \rangle$. In particular, this method relies on the Central Limit Theorem, which assumes that the distribution of independent random variables, in our case the errors, converges to a standard normal distribution. The iterative algorithm used to model the error can be summarized in the following 4 steps, with a user-defined refresh rate $T \in \mathbb{N}$ and a maximum number of simulated samples N_{max}:

1. Create an initial population P consisting of 30 random input values $\langle i, j \rangle$ for the circuit.

2. Compute μ_{est} and f_{est} over P, and their standard deviations σ_μ and σ_f.

3. From μ_{est}, f_{est}, σ_μ, σ_f, and from confidence constraints h and p, infer $N_{min} \in \mathbb{N}$, the estimated minimum number of samples to simulate.

4. If $|P| < N_{min} < N_{max}$, add T new random input values to P and go back to step 2. *Otherwise*, enough samples have been evaluated, and μ_{est} and f_{est} satisfy the requested confidence constraints.

Thus, based on the empirical values of μ and f computed on a subset of the inputs, the algorithm automatically infers the remaining number of input values N_{min} to simulate to satisfy its constraints. Since inferential statistics are used in step 3, it follows that the larger the estimated standard deviation σ_μ and the tighter the confidence constraints, the larger N_{min} will be. Nevertheless, by re-evaluating N_{min} every T new input pairs, the algorithm converges to a minimized value for N_{min}.

Experimental results

The CASSIS method was evaluated on 8 different arithmetic circuits, with operand word lengths ranging from 8-bit to 32-bit [C6]. All experiments were performed with $h = 5\%$ and $p = 95\%$, and $N_{max} = 2.5 * 10^7$. Only the most important experimental results are presented in this section, for a detailed presentation of the experiments the interested reader should refer to [C6].

Fast Convergence: When it converges automatically, that is, for operators with a small standard deviation of errors, the CASSIS method simulates a very limited number of input values. On average, only 25% of all values are simulated for 8-bit operators, $1.5 * 10^{-3}\%$ for 16-bit operators, and $2.4 * 10^{-11}\%$ for 32-bit operators. In all cases, at most a few minutes of simulation are required to run the proposed algorithm.

Adaptive Complexity: While it may be tempting to run an arbitrarily large number of simulations to predict the error models, for example 5M pairs of inputs, the results show that for some operators this number is sometimes far too small, while for others it is far too large. For example, experiments show that for various operators, between 50k and 17M input pairs are needed to characterize their error with confidence constraints of $h = 5\%$ and $p = 95\%$.

3.2.2 System-Level Error Characterization with Kriging

Simulations for Approximate DSE

Characterizing the impact of errors at the system level is a key step in the DSE process. While at the operator level the goal of error characterization is to derive a statistical model of errors for a fixed logic component, at the system level the challenge is to evaluate many potential approximate configurations to identify the most appropriate one. A simple example of such a DSE process is the selection of the appropriate fixed-point format for n different data channels within a complex system. Assuming an 8-bit constraint on data formats, 9 fixed-point configurations can be used for each data channel, for a total of 9^n system configurations. Thus, for a system with only 5 data channels, 59049 configurations are possible.

In complex stream processing systems that mix complex control and data streams, the impact of approximations on system performance is not trivial to model. As with operator-level error characterization, simulations must be used to evaluate the impact of a given approximation configuration on system performance. For each considered configuration, the system behavior needs to be emulated with a representative input data set in order to generate a quality metric for that approximate configuration [C8].

Simulating a complex algorithm is time-consuming, especially when it is necessary to emulate approximations within the algorithm. The total DSE time T_{DSE} can be modeled by the following equation:

$$T_{DSE} = Nb_{config} \times T_{sim} \times Nb_{sim}$$

where Nb_{config} represents the number of configurations evaluated, T_{sim} the time taken by each simulation, and Nb_{sim} the number of simulations needed to produce a relevant system quality metric for the DSE process. Many state-of-the-art techniques focus on speeding up the DSE process by reducing Nb_{sim}, using heuristics to find the most promising configurations faster [P1, 74]. Another common approach to speed up the DSE process is to reduce the simulation time T_{sim}, for example by using statistical models of approximate operators instead of bit-accurate exact simulations [C11].

Kriging-based DSE acceleration

The goal of the contribution proposed in [C9] is to speed up the DSE process of approximate systems by skipping the simulation of some explored configurations and predicting their results instead. The basic idea behind this technique is that a configuration simulation that produces a quality metric may be skipped if enough nearby configurations have already been simulated. In such a case, the quality metric for the evaluated configuration is predicted using an interpolation of its neighbor evaluations, which is much faster to compute than simulating the system.

The identification of skippable simulations and the interpolation methods are both based on the Kriging technique [18]. Kriging is a technique used in geostatistics, and more specifically in mining, to predict the value of random multidimensional functions from known nearby values, with a confidence interval. Without going into details, from a set of known and spatially close values of a random function, the Kriging technique identifies a so-called semi-variogram that relates these known values to each other. The identified semi-variogram can then be used to predict the value of the random function by interpolation, with a confidence interval depending on the proximity to the set of known values. In practice, each configurable approximation implemented within a system represents a dimension of the explored space. For example, each variable with a configurable word-length format, or each operator with a tunable precision, represents a dimension for the Kriging technique.

Experimental results

The Kriging-based method for accelerating the DSE of the approximate system has been implemented for 5 applications, ranging from simple FIR (Finite Impulse Response) filters to part of the HEVC (High-Efficiency Video Coding) video encoder and the SqueezeNet CNN (Convolutional Neural Network) [C9]. For digital signal processing applications, the quality metric estimated by the technique is the noise power at the output of the system, while for the CNN the more complex rate of successful classification is evaluated, demonstrating the genericity of the proposed method.

Depending on the confidence interval imposed for the Kriging interpolation method, on average between 65% and 86% of the configurations are interpolated instead of simulated. Thus, the confidence interval imposed by the user controls the trade-off between the number of interpolated simulations and the accuracy of those interpolations. Overall, this technique can reduce the time required for DSE by a factor of 2 to 10, with larger gains for the largest applications, even with fewer dimensions to explore. More detailed experimental results can be found in [C9, 11].

3.3 Lossy Image Compression for Machine-to-Machine Communications

Approximate techniques presented in Section 3.2 are generic error characterization techniques applicable to any circuit or system with well-defined quality metrics and configurable approximations. Rather than focusing on the search for generic methods, this section studies the impact and resilience to approximations for a specific application context: machine-to-machine communications, and more specifically Video Coding for Machines (VCM). Section 3.3.1 introduces the challenges of the VCM context, and Sections 3.3.2 and 3.3.3 present contributions in this area.

3.3.1 Challenges and Opportunities for Machine-to-Machine Image Communications in the AI Era

The IoT era has accelerated the trend toward a society in which a distributed network of embedded systems and intelligent sensors communicate with the cloud to provide services. Combined with the leap in capabilities of AI techniques, the forthcoming ubiquity of IoT devices paves the way for applications such as smart cities, smart buildings, and e-health [33].

In a typical sensor network system, a number of low-power IoT devices generate data that is transmitted to a dedicated facility, such as the cloud or fog computing, for processing. Cisco estimated that between 2016 and 2021, data generated by IoT devices and transmitted to data centers for processing would grow from 220 to 847 Zetabytes [17]. In this context, a key challenge for the AI and signal processing scientific community is to find innovative solutions to contain this so-called *deluge* of data from IoT devices. It is therefore necessary to consider aggressive data compression solutions to limit traffic bandwidth and reduce the energy footprint associated with IoT transmissions, while preserving the benefits of massive IoT deployment.

To reduce the amount of data transmitted, an opportunity lies in the ephemeral nature of 99% of the data generated by IoT devices [17]. Ephemeral data is data that is transmitted to a processing facility where it is used directly to provide a service, but is not stored for further reuse. This ephemeral nature of data creates an opportunity for approximation in the form of aggressive lossy data compression techniques. Lossy compression consists in reducing the volume of transmitted data by voluntarily discarding useless parts of the transmitted information, thus lowering its intrinsic quality. In fact, since ephemeral data is neither stored nor reused, its quality can be reduced to a bare minimum, just high enough to allow correct interpretation by AI techniques.

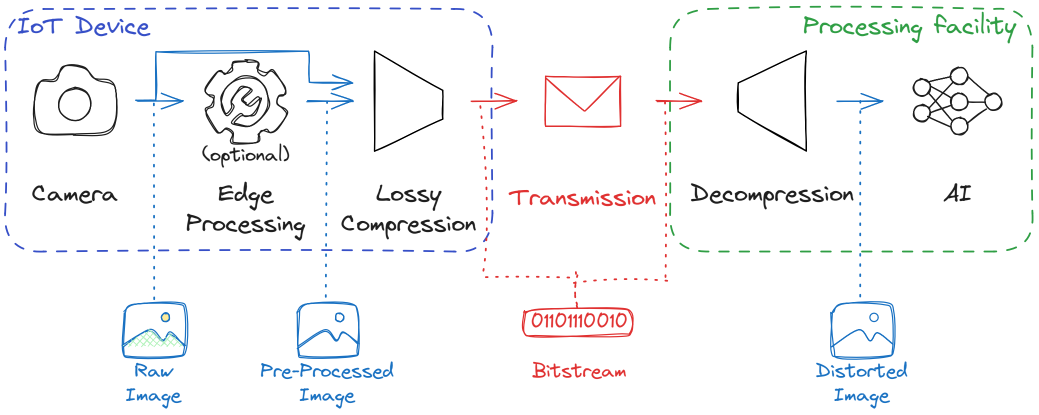


Figure 3.3: Image and Video Coding for Machine Information Flow

The work presented herein focuses on the transmission of approximated image and video data, degraded by lossy compression techniques, and consumed by an AI-based computer vision algorithm. A typical flow for such a VCM system is presented in Figure 3.3. On the edge or IoT device side, the image captured by the camera may be subject to local processing before transmission, which is often limited by the small computational capacity of embedded processors, especially on battery-powered devices. This local processing can be used, for example, to perform a lightweight first part of the computer vision task, or to pre-process images before compression. For example, image preprocessing is used in [96] to identify potential regions of interest in images captured by a drone, and to remove chrominance components for all pixels outside that region before transmission. On the processing facility side, which is usually located close to the device in fog computing or in cloud computing facilities, processing power is generally not an issue, and power-hungry AI are usually used to process the data.

The challenge in designing a VCM system like the one presented in Figure 3.3 is to optimize and balance four aspects:

- the compressed image size, which affects the bandwidth needed for transmission,
- the embedded computational complexity, latency, and throughput for compression and pre-processing, which must match the capabilities of the embedded device and the real-time requirements of the applications,
- the power consumption associated with compression and transmission, which is critical for battery-powered IoT devices,
- the AI quality of service, which must be maintained at an acceptable level for the end user of the system.

The contribution presented in Section 3.3.2 focuses on the AI part of the processing facility, studying its resilience to lossy compression artifacts. Then Section 3.3.3 outlines a study of the impact of various lossy compression schemes from the device side on various computer vision tasks.

3.3.2 Enhancing AI Resilience to Lossy Compression Artifacts

The goal of this contribution is to improve the resilience of a given CNN to image degradation caused by a particular lossy encoding scheme. The different strategies that can be used to train a CNN when considering images degraded by a lossy image encoder are illustrated in Figure 3.4.

In most AI-based computer vision applications, image compression effects are simply ignored, and CNNs are trained on supposedly pristine images. As shown in the original training method in Figure 3.4, the weights of the CNN are simply trained using a backpropagation of prediction errors given by a loss function that compares the network predictions to a reference gold truth (not shown in the figure). The main problem with

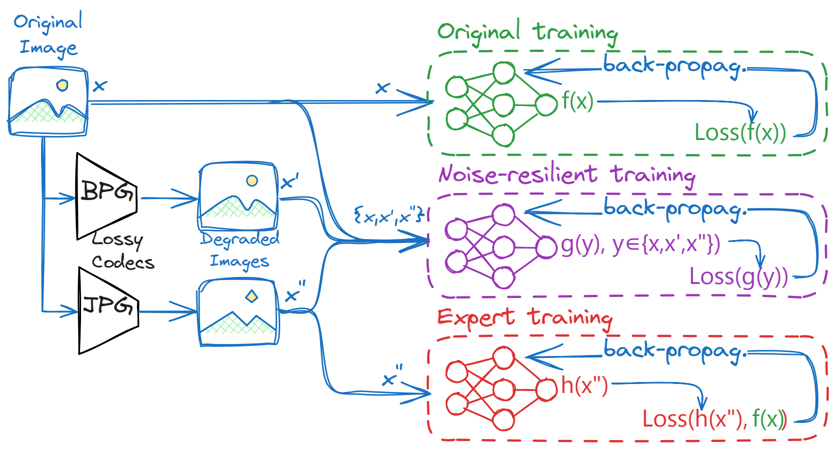


Figure 3.4: Expert training for enhancing resilience of CNNs to compression noise.

this classical technique is that the performance of the trained CNNs can degrade rapidly in the presence of noise not seen during the training process [C38, T2, 19, 22].

From the literature, a first way to train noise-resistant CNNs is to expose them to multiple types of noise during the training process. While this technique provides good results, the potential caveats are an unwanted drop in accuracy on pristine images and still significantly lower performance on distorted images [112].

The proposed contribution [C37] is a training technique aimed at making a network expert for a very specific type of noise. The intuition behind this idea is that when designing a VCM system, the designer will choose a specific image processing system with a well-defined coding algorithm. Thus, instead of trying to be resilient to many types of noise, expert training focuses on a single type of noise, and does not even consider pristine images for training the CNN.

Expert training of a network $h(.)$ goes beyond simple fine-tuning on degraded images of a network $f(.)$ trained on pristine images, as illustrated in Figure 3.4. A key element of the proposed technique, inspired by [112], is that the training of $h(.)$ on a degraded image x'' is boosted by forcing it to provide a similar result to $f(.)$ exposed to the original undegraded image x.

The expert training technique was evaluated on two image classification CNNs, namely MnasNet and ResNet50, trained on the ImageNet dataset [53]. Three image encoders, namely JPEG, JPEG2000, and BPG, were used in the experiments, each with three degrees of distortion. As expected, specializing a network for a specific type and degree of distortion significantly increases its accuracy for that configuration. On average, the expert network is able to increase the classification accuracy by 2% to 45% compared to a network trained only on pristine images. Compared to a fine-tuning technique on the selected encoder and distortion level, the proposed expert training technique also achieves an average 0.61% better classification accuracy. A detailed experimental evaluation of the expert training technique is presented in [C37], in particular an analysis of the noise resilience of different encoders for expert networks.

3.3.3 Towards Exploitation of the Resilience of AIs to Lossy Compression to Lower Compressed Image Bitrates

This contribution studies the impact of a wide set of lossy encoders and lightweight preprocessings on the performance of an AI-based computer vision algorithm. The main goal of this study is to find, for each encoder, the set of Pareto configurations that give the best achievable tradeoffs between the size of the transmitted images (*i.e.* the bitrate) and the accuracy of the AIs. For a fair comparison, the expert training presented in Section 3.3.2 is used in each considered configuration to ensure optimal performance of the AI.

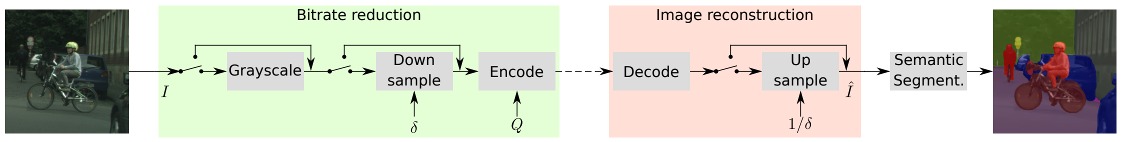


Figure 3.5: Evaluation pipeline for encoder in a VCM context (from [C38]).

The evaluation pipeline used in this contribution is illustrated in Figure 3.5. Two optional preprocessings of the input image can be used in this study: a grayscale image conversion, which consists of ignoring the color components of the image, and a downsampling of the input images to a lower resolution using a bicubic interpolation algorithm. In addition to this optional preprocessing, the quality parameter Q of the image encoders can be configured to different values, each offering a different trade-off between quality and bitstream size. The AI task considered in this study is image segmentation, which aims to delineate and classify objects within an image.

During this study, 5 standard image and video encoders were considered: JPEG, JPEG2000, JM, X265, VVenc. It should be noted that the video encoders were used in intra-mode to encode the still image under consideration, thus not using temporal prediction mechanisms that are useful for video compression. For each encoder, up to 7 image resolutions and up to 11 quality parameter values were tested, in addition to the optional grayscale pre-processing. In total, 1486 different encoding and pre-processing configurations were evaluated in this study.

Since training a deep learning network for each configuration is time-consuming, a faster training strategy, named progressive training, is proposed in [C38]. The progressive training strategy consists of iteratively increasing the values of the downscaling and quality parameters, and fine-tuning the network weights along the way, instead of starting training from scratch for each configuration. The speed of exploration of configurations can be traded off against the accuracy of the trained networks using a parameter s to control the speed of iterative progress of quality and scaling values. This method can save between 74% and 13% of the training time for exploring all configurations of a given encoder, compared to training dedicated networks for each configuration. The impact of progressive training on the accuracy of the image segmentation network is minimal, ranging from a loss of 16% to a gain of 2.62%, for faster and slower values of s, respectively.

A first interesting counterintuitive result is that removing color from images does not result in interesting tradeoffs between image size and accuracy in most of the newer encoders: JM, x265, and VVEnc. Although this is a strategy used in some publications [96], this result is explained by the fact that color information is only a small part of the encoded bitstream, and that removing color strongly affects the accuracy of deep learning networks. For older encoders, where color is still a larger part of the bitstream, the grayscale option may be interesting for the lowest quality configurations.

A second interesting result of this study is that image resolution plays an important role in the tradeoff between accuracy and bitstream size. To show this, two sets of configurations are compared: on the one hand, configurations obtained by varying only the quality parameter of the encoder and, on the other hand, by also varying the downscaling factor. The Pareto fronts of the tradeoffs produced by the two configurations, in the accuracy and bitrate 2D space, are compared using the Bjøntegaard-Delta Rate (BDR) metric. The BDR metric represents the average bitrate savings at equivalent network accuracy between two sets of configurations. Using this approach, JPEG achieves 43% bitrate savings over JM, which in turn saves 14% over x265, which saves 19.9% over VVenc. Therefore, older encoders, which are several orders of magnitude less complex than most newer ones, can save bandwidth by downsampling if lower network accuracy is acceptable. This can be explained by the fact that most recent encoders are more suitable for larger images, beyond 720p resolutions, but do not perform well for smaller images. This is due to the block partitioning mechanism implemented in recent codecs, which cannot be applied when the compressed image size is close to or below the minimum size of a basic block.

A conclusion of this study, supported experimentally [C38], is that finding the best encoding configuration is not straightforward when considering the complexity of the encoder, the use of lightweight pre-processing, different encoders, different hyper-parameters for system configuration, and different quality and bandwidth constraints.

3.4 Summary

In this chapter, contributions to approximate computing techniques have been reviewed. These contributions include domain-agnostic methods and tools to support the DSE of approximate computing systems, and more specific contributions that exploit and study the resilience of AI to compression approximation in the context of VCM.

The contributions presented here were made possible by the hard work and dedication of the following individuals:

- **PhD students**: Justine Bonnot, Alban Marie.
- **Master students**: Mathieu Dejean-Servières.

Research perspectives on approximate computing techniques include work on transpilers for automated use of approximation at the source code level, and on automation of optimization of VCM systems. These axes are described in detail in Chapter 5.

Hoooodor. Hoooooooodor, hoooooodor, hoDOR, hoDOR, hoDOR.

Hodor, to himself,
A Storm of Swords,
George R.R. Martin

Chapter 4

Low-Complexity Reinforcement Learning with Tangled Program Graphs

In less than a decade, AIs powered by Deep Neural Network (DNN) have outperformed and replaced human-made algorithms in many application domains, from computer vision [13] to Natural Language Processing (NLP) [1]. This sudden breakthrough of Deep Neural Networks (DNNs) is largely due to the availability of affordable and easily programmable hardware with significant computing power, such as GPUs [53]. Driven by the ever-increasing computing power of commercial chips, the current race for omnipotent AIs is leading to the creation of increasingly complex DNNs requiring millions [13] to hundreds of billions [1] of parameters.

Contrary to the increasing computational complexity of DNNs, the need for lightweight AIs is also growing. In fact, the ubiquity of IoT devices and the enormous amount of data they generate [17], call for new paradigms where data processing is performed locally, close to the data producer. The processing of data by AIs on embedded IoT devices is not compatible with the use of compute-, storage- and power-hungry DNNs.

A common way to create lightweight and frugal AIs is to exploit the resilience of DNNs to approximations, using techniques such as pruning, low-precision, and approximate computation [72, 102]. Using such techniques, it is possible to simplify the trained DNN models while retaining most of their accuracy, with *typical* gains of up to 1 order of magnitude in computational complexity and 2 orders of magnitude in memory usage [72, 102]. Even with these techniques, only relatively small and less powerful DNNs, such as MobileNet or EfficientNet [99], can be embedded in tight memory-, energy-, and time-constrained systems.

An alternative way to create lightweight AIs is to develop new machine learning techniques that rely on light-by-construction models, such as the TPG model studied in this chapter. The chapter is structured as follows: Section 4.1 introduces the TPG technique for constructing ultra-light Reinforcement Learning (RL) AIs. Then, Sections 4.2 and 4.3 present contributions for accelerating the training and inference of TPG-based AIs. Some additional contributions using TPGs are outlined in Section 4.4. Finally, Section 4.5 concludes the chapter.

This chapter summarizes contributions and gathers content from the following personal publications: [J14, C12, C17, C23, C46, T1].

4.1 Introduction to Tangled Program Graphs

TPG, which stands for Tangled Program Graph, is a Reinforcement Learning (RL) technique proposed by Stephen Kelly and Malcolm Heywood in [47]. Building on state-of-the-art genetic programming techniques, TPGs are grown from scratch for each learning environment in which they are trained. The content of this section is also covered in a 9-minute video introduction to TPGs produced in 2020 [T3].

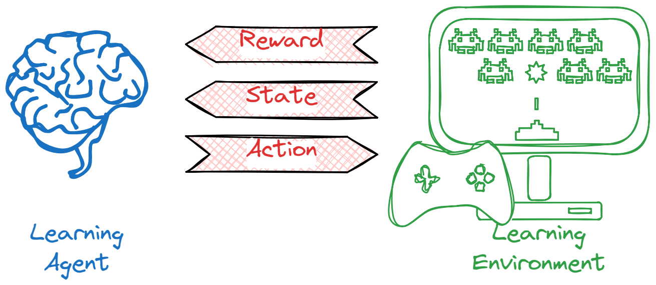


Figure 4.1: Reinforcement Learning (RL) principles

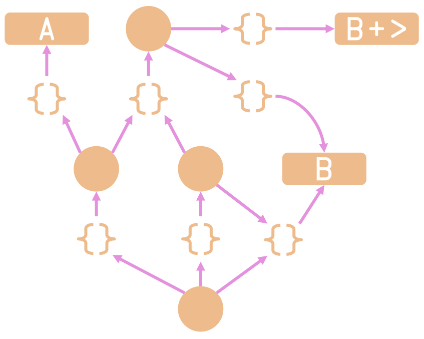
Background: Reinforcement Learning and Genetic Programming

Reinforcement Learning is a branch of machine learning techniques in which an artificial intelligence learns how to interact with an environment through trial and error, as illustrated in Figure 4.1. In RL, an artificial intelligence, called a learning agent, observes the current state of its learning environment and interacts with it through a finite set of actions. As a result of these actions, or due to external phenomena such as time or physics, the state of the learning environment evolves. By observing the constantly evolving state of the environment, the learning agent has the opportunity to react and build a meaningful sequence of actions. In order for the agent to learn which sequences of actions are useful, an additional reward mechanism is implemented. By rewarding useful behavior of the learning agent and penalizing harmful or useless behavior, this reward mechanism helps the learning agent select the most appropriate behavior for each new experience. Although TPGs were originally developed for RL purposes, the possibility of adapting them to other types of learning environments, such as classification or time series predictions, has already been demonstrated [48].

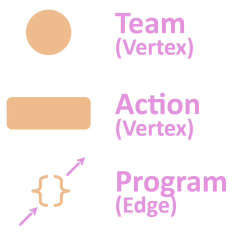
Genetic programming is a subset of machine learning techniques that mimics the evolutionary process of natural selection to breed programs for a selected purpose [113]. The iterative learning process of genetic programming can be broken down into four steps:

1. Create an initial population of $n \in \mathbb{N}^*$ random programs.
2. Evaluate the fitness of the n programs against the learning environment.
3. Discard the $m < n, m \in \mathbb{N}^*$ programs in the population with the worst fitness.
4. Regenerate m new programs from the remaining programs using genetic operations such as mutations or crossovers.
5. If a termination criterion, usually a maximum number of generations, is not met, go back to step 2 to iterate the natural selection process.

As detailed in [46, 47], TPGs add a compositional mechanism to this genetic learning process that favors the emergence of stable clusters of useful programs by building a hierarchical decision structure.



(a) TPG example



(b) TPG semantics

Figure 4.2: Semantics of the Tangled Program Graphs (TPGs)

TPG: Model Semantics

The semantics of the Tangled Program Graph (TPG) model, shown in Figure 4.2, consists of three elements that form a direct graph: *programs*, *teams*, and *actions*. The *teams* and *actions* are the vertices of the graph, *teams* are the internal vertices and *actions* are the leaves of the graph. The *programs*, associated to the edges of the graph, each connecting a source *team* to a destination *team* or *action* vertex. Self-loops, that is, an edge connecting a *team* to itself, are not allowed in TPGs.

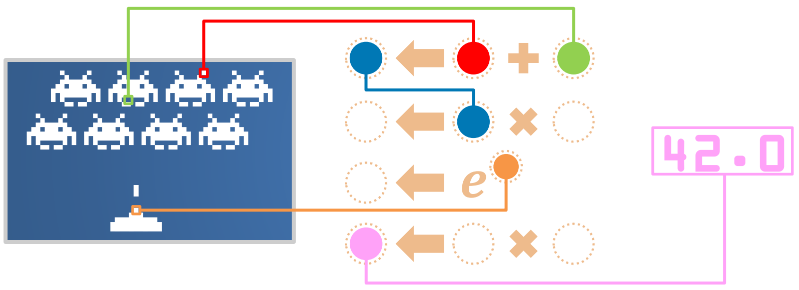


Figure 4.3: *Program* from a TPG. On the left, the learning environment state fed to the *program*. In the middle, the sequence of instructions of the *program*. On the right, the result produced by the program.

From afar, a *program* can be seen as a black box that takes the current state of the learning environment as input, processes it, and produces a real number, called a *bid*, as a result. More concretely, a *program* is a sequence of simple arithmetic *instructions*, such as additions or exponents. As shown in Figure 4.3, each *instruction* takes as operand either data coming from the observed learning environment, or the value stored in a register by a previous *instruction*. The last value stored in a specific register, generally called `R0`, is the result produced by the *program*.

The execution of a TPG starts from its unique root *team* when a new state of the environment becomes available. All *programs* associated with outgoing edges of the root *team* are executed with the current state of the environment as their input. Once all *programs* have completed their execution, the edge associated with the largest *bid* is identified, and the execution of the TPG continues following that edge. If another *team* is pointed by this edge, its outgoing *programs* are executed, still with the same input state, and execution continues along the edge with the largest *bid*[1]. Eventually, the edge with the largest *bid* leads to an *action* vertex. In this case, the *action* is executed by

[1]Although theorized and supported in [46], cyclic paths are never structurally introduced in trained TPGs.

the learning agent, a new resulting state of the environment is received, and the TPG execution restarts from its root *team*.

TPG: Training Process

The genetic evolution process of a TPG is based on a graph with several root *teams*. The initial TPG created for the first generation contains only root *teams* whose outgoing edges each lead directly to a *action* vertex. At a given generation of the learning process, each root *team* of the TPG represents a different *policy* whose fitness is evaluated. Evaluating a root *team* consists of executing the TPG stemming from it a fixed number of times, or until a terminal state of the learning environment is reached, like a game-over in a video game. The rewards obtained after evaluating each root *team* of the TPG are used by the genetic evolution process. The least fitting root *teams* with the lowest rewards are deleted from the TPG.

To create new root *teams* for the next generation of the evolution process, randomly selected remaining *teams* from the TPG are duplicated with all their outgoing edges. Then these new edges undergo a random mutation process, possibly changing their target vertex and modifying their *programs* by adding, removing, swapping, and changing their instructions and operands. Surviving root *teams* from previous generations may become the destination of an edge added during the mutation process, thus becoming internal vertices of the TPG. This mutation mechanism favors the emergence of long-lived, valuable subgraphs of connected *teams*. Indeed, useful *teams* that contribute to higher rewards have a greater chance of becoming internal vertices of the TPG, which cannot be discarded unless they become root *teams* again. Thus, complexity is adaptively added to the TPG only if that complexity leads to better rewards for the learning agent. A detailed description of this evolution process can be found in [46].

TPG Capabilities

The capabilities of TPGs have been extensively demonstrated [46, 47] on the 55 video games of the Arcade Learning Environment (ALE) [7]. In this learning environment, the adaptive complexity leads to TPG of different sizes, depending on the complexity of the strategies developed to play each game. For example, there are two orders of magnitude between the number of vertices of the smallest and largest networks built in these learning environments. On the performance side, TPGs has been shown to achieve a level of competence comparable to state-of-the-art deep learning techniques on ALE games, at a fraction of their computational and memory cost. Compared to state-of-the-art techniques, TPGs achieve comparable competency with one to three orders of magnitude fewer computations, and two to ten orders of magnitude less memory required to store their inference model [46, 94]. Recently, an extension of the TPG model supporting continuous action space has been proposed to address new learning environments, such as time-series prediction [48].

4.2 Acceleration of Tangled Program Graph Training with Deterministic Parallelization

GEGELATI[2] is an open source framework, developed at IETR, for training and execution of TPGs [C23]. The main motivation behind the creation of the GEGELATI library is the desire to have an efficient, embeddable, portable, parallel and deterministic library. Because of the efficiency and embeddability goals, C++ was a natural choice for the development of GEGELATI. Previous open source C++ implementations, including the reference C++ code from Kelly [47], were neither parallel nor deterministic. The creation of a new library from scratch was also motivated by the low code quality of existing C++ implementations, particularly due to a lack of code documentation and monolithic code.

From the beginning, the GEGELATI library has been designed to promote its adaptability to different learning environments and its portability to different architectures, without sacrificing its performance. To this end, two original contributions have been integrated into the library: the parallelization of the deterministic learning process, presented below; and the support for customizable *instructions*, detailed in [C23].

4.2.1 Motivations for Deterministic Parallelism in TPG training.

The portability of the GEGELATI library allows it to be used on both general-purpose and embedded architectures. Indeed, when training a learning agent to run on an embedded system, a common design process is to first prototype the agent on a general-purpose processor before embedding it on the embedded target. Portability also makes it possible to train a learning agent offline on a HPC architecture before deploying it for inference on a less powerful architecture.

Parallelism in the learning process is an essential feature for accelerating the training of new learning agents, which encourages the adoption of a new machine learning technique. Indeed, the breakthrough of deep learning models is largely due to the acceleration of their training process with GPUs [53]. Support for parallel computation is

[2]GEGELATI (Generic Evolvable Graphs for Efficient Learning of Artificial Tangled Intelligence)

useful for general-purpose and HPC architectures, but also for embedded systems, which today often integrate heterogeneous MPSoCs.

Determinism of a learning process is the property that ensures that, given a set of initial conditions, the learning process will always end with the same result. Determinism can only be obtained under the assumption that the state of the learning environment itself changes deterministically, depending only on the sequence of actions applied to it. Determinism is a key feature, especially for a pseudo-stochastic learning process like the training of TPGs. In fact, the outcome of training may depend in part on luck, which is precisely why being able to reproduce a result deterministically is crucial.

Determinism is antagonistic to the goals of parallelism and portability, and to the stochastic nature of the learning process, making it difficult to achieve all of these goals together. In fact, parallelism is inherently a source of non-determinism, since the concurrency of computations accessing and modifying shared resources, often in an unknown order, tends to produce variable results.

4.2.2 Deterministic and Scalable Parallel TPG training

During the learning process of TPGs, the most computationally intensive parts are the fitness evaluation of the *policies* and the mutations of the *programs* added during the evolution process.

The fitness evaluation of individual *policies* can be performed deterministically in parallel, provided that 1/ the learning environment can be cloned to evaluate multiple policies simultaneously, and 2/ any stochastic evolution of the state of the learning environment can be controlled deterministically. Under these conditions, the parallel evaluation of *policies* is possible because the topology of the TPG, which is a shared resource for all *policies*, is fixed during this evaluation process.

The mutation of *programs* can be applied deterministically in parallel. Two types of mutations are applied to the TPG: mutations that affect the graph topology by inserting new root *teams* and edges; and mutations that affect the *instructions* of the *programs* associated with the new edges. While mutating the graph topology cannot be done in parallel, since the graph is a shared resource, individual *programs* are independent and can be mutated in parallel.

To control a stochastic process, a Pseudo-Random Number Generator (PRNG) must be used each time a random number is needed. Given an initial seed, a PRNG produces a deterministic sequence of numbers. Such pseudo-random numbers are needed in the TPG mutation process, but can also be used to simulate stochastic behavior in a learning environment. To ensure full determinacy of the training of a TPG, a unique PRNG should be called in a fixed order during the whole training. It is not possible to let the parallel parts of the training process call the PRNG directly, because the absolute order in which the parallel computations occur is itself random. It is also not possible to give each parallel task a precomputed list of pseudorandom numbers, since the number of random numbers needed for each task is itself stochastic. For example, when mutating a *program*, mutations are applied iteratively until the behavior of the program becomes "original" compared to pre-existing *programs* in the TPG. Therefore, it is not feasible to specify a fixed number of precomputed random numbers for the *program* mutations.

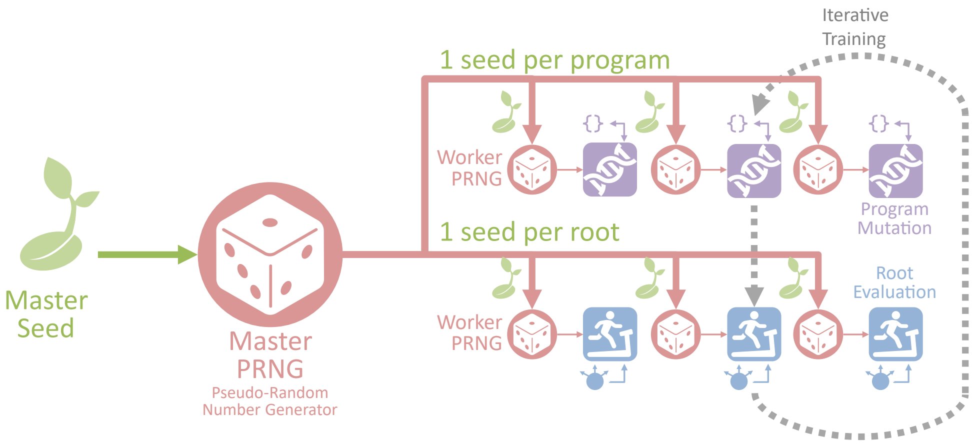


Figure 4.4: Deterministic and parallel TPG training.

The parallelization strategy adopted in Gegelati is based on the master/worker principle, with a distributed PRNG. The principle of the distributed PRNG, illustrated in Figure 4.4, is the use of two different PRNG instances: the $prng_{master}$ and the $prng_{worker}$. The $prng_{master}$ is used exclusively in the sequential parts of the learning process, which gives it a deterministic nature given an initial seed. Besides being used for stochastic tasks performed sequentially, such as TPG topology mutations, the $prng_{master}$ is also used to generate a seed for each parallel worker task. In each worker task, a private $prng_{worker}$ is instantiated and initialized with the seed provided by the $prng_{master}$. Since all calls to PRNG from worker tasks use only their private $prng_{worker}$, the random number sequences generated in each parallel task are deterministic.

The detailed pseudocode of the master and worker tasks for policy fitness evaluation is presented in [C23], and an open source implementation is available on GitHub[3].

[3] Gegelati. GitHub repository: `http://github.com/gegelati/gegelati`

4.2.3 Evaluation of TPG Training Parallelization

The proposed parallelization strategy for training TPGs was evaluated on several targets, including a high-end *Intel* Xeon 24-core CPU, a laptop *Intel* i7 8-core CPU, and an embedded *Arm* Big.LITTLE 8-core CPU.

The first important result obtained is that, despite its additional features, the sequential performance of GEGELATI for training TPGs is on average 11% faster than the reference code of Kelly et al. [47]. This result was not a foregone conclusion, as the instruction set customization and determinism enforced by GEGELATI have a non-negligible performance overhead.

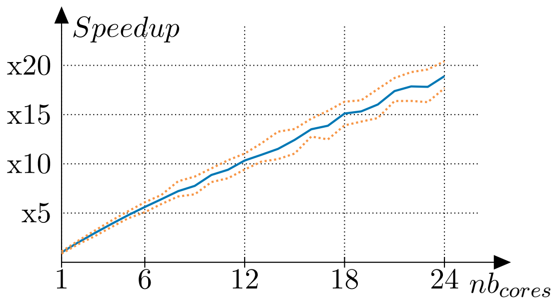


Figure 4.5: Speedup of TPG training on an *Intel* Xeon E5-2690 24-core CPU.

To evaluate the scalability of the parallelism on each target, TPGs were trained in 5 different learning environments from the ALE, and an inverted pendulum for training run on the embedded ARM processor. The training time was measured for a number of threads varying between 1 and the number of physical cores of the architecture. No affinity constraint was imposed on the threads, leaving the scheduling to the Linux operating system. Each experiment with a chosen environment and number of threads was repeated 5 times with different seeds for the master PRNG.

Figure 4.5 shows the speedup results, compared to sequential training, for the *Intel* Xeon 24-core CPU. The thick line represents the average speedup observed for all learning environments and seeds, and the dotted lines represent the minimum and maximum speedups observed. As can be seen from these results, parallelizing the learning process is highly scalable, with an average speedup of 18.9 on the 24-core Xeon CPU. Similar results were obtained on other targets, with a speedup of 2.61 on the *Intel* i7 4-core CPU and a speedup of 4.12 on the *Arm* Big.LITTLE CPU when using the 4 large A15 cores and the 4 smaller A7 cores simultaneously.

4.3 Ultra-Fast Inference of Tangled Program Graphs through C Code Generation

This section introduces a new design flow to enable ultra-fast and lightweight inference of RL agents based on the TPG model. The core of this contribution consists of translating pre-trained TPG graphs into standalone and standard C code that can be compiled without relying on any third-party library.

4.3.1 Motivations and Opportunities for Inference Acceleration

The main motivation behind this contribution is to accelerate the inference of TPGs by reducing the computational complexity of the inference process. A first benefit of accelerating the inference of TPGs is to obtain better response time for AI agents implemented with this model, which can be a strong asset for integration in tightly constrained real-time systems. A second benefit of faster inference time is energy savings. Indeed, reducing the computation time required to obtain the same result reduces the computational load on the processor used, which generally translates into power savings. A third benefit of reducing the computational complexity of TPG inference is that it may allow developers to meet their time constraints while using less powerful, less expensive, and less polluting hardware for their systems.

This contribution focuses on accelerating the inference of TPGs whose graph topology and *programs* have been fixed during a training process. Considering a TPG with a fixed topology provides many opportunities for acceleration:

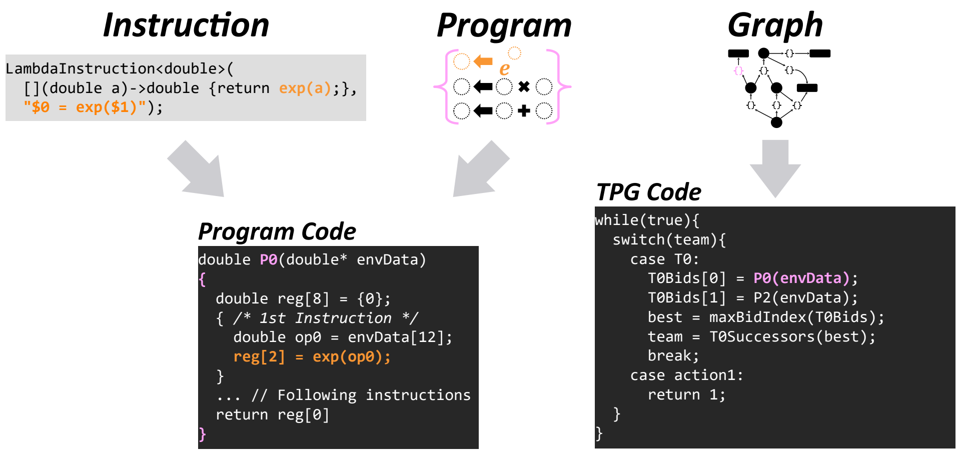
- *Removing overhead for dynamic TPG structures.* During the training process, the topology of the TPG *team*, its number of *programs*, and the *instruction* list of each *program* may change as a result of the genetic mutation process. To support these constant mutations, flexible data structures must be used to support the creation and destruction of random *teams*, *programs*, and *instructions*. The flexibility of these dynamic graph and list data structures, which imposes memory and computation overhead, is not needed for inference and can thus be optimized.

- *Removing TPG instruction decoding and data fetching overhead.* Because during training, TPG *programs* contain a dynamic list of *instructions* that refer to operands that may also mutate during the genetic evolution process, executing these programs requires software *instruction* decoding and data fetching mechanisms. Even in their simplest forms, such as using function pointers for *instructions* and indexed arrays for data, these decoding and fetching mechanisms introduce indirection overhead during program execution. In a program with a fixed list of *instructions* with known operands, these software instruction decoding and data fetching mechanisms are no longer needed.

- *Getting rid of complex TPG framework.* While the training of TPGs requires substantial software to handle the pseudo-random mutations, export and import of TPG files, or the parameterization and logging of the training process, none of these features are needed to infer a pre-trained TPG. The benefits of discarding useless software are: 1. reduce the memory footprint of the software, which is critical on embedded targets, and 2. avoid having to build complex, useless third-party dependencies to support TPG inference.

- *Benefiting from compiler optimization.* By translating a pre-trained TPG into standalone standard C code, the inner workings of the TPG and its *programs* are directly exposed to C compilers, making it possible to benefit from their powerful optimization passes.

4.3.2 C Code Generation for TPG Inference



```
LambdaInstruction<double>(
  [](double a)->double {return exp(a);},
  "$0 = exp($1)");
```

```
double P0(double* envData)
{
  double reg[8] = {0};
  { /* 1st Instruction */
    double op0 = envData[12];
    reg[2] = exp(op0);
  }
  ... // Following instructions
  return reg[0]
}
```

```
while(true){
  switch(team){
    case T0:
      T0Bids[0] = P0(envData);
      T0Bids[1] = P2(envData);
      best = maxBidIndex(T0Bids);
      team = T0Successors(best);
      break;
    case action1:
      return 1;
  }
}
```

Figure 4.6: Principles of C Code Generation for TPG inference

The various parts of the proposed code generation framework [C17], implemented within the GEGELATI library, are illustrated in Figure 4.6. Two main pieces of code are generated by this process: *Program* code and *TPG* code.

Program Code Generation

In a trained TPG, a program is an evolved sequence of assembly-like instructions that process the data observed by the learning environment to produce a single `double` value as a result. To generate the C code corresponding to a *program*, each *instruction* must be translated into C code. To support custom instructions declared by the user of the library, a string template is used when declaring the *instruction*, as shown in Figure 4.6. This string template, which adopts the syntax of regular expressions, uses the placeholder `$0` for the register that stores the result returned by the instruction, and the placeholder `$n` for the name of the n^{th} operand of the instruction.

For each *program* of the TPG, a separate C function is printed, as shown in Figure 4.6. The printed function takes as arguments the pointers to the data sources used to observe the current state of the environment. Then the function declares the registers used to store the results of the *instructions* throughout the *program*. The *instructions* of the *programs* follow and are printed one by one. For each instruction, the operands are first retrieved from the environment data source. For simple data types this is done by simple pointer dereferencing, as shown in Figure 4.6. For more complex operand types, such as arrays of data, the container class managing the environment data may provide more complex code generation schemes for fetching the operands. Finally, the value held in the first register is returned as the *bid* for the *program*.

TPG Code Generation

The TPG structure is encoded as an Finite State Machine (FSM) using a `switch-case` structure. A snippet of the TPG switch structure is shown in Figure 4.6. Each `case` represents a *team* containing several *program* executions. The transitions in the FSM represent the edges of the graph. The results of *programs* are stored in a special array to avoid running *programs* multiple times, which can happen when a program is used by several teams. Traversing a leaf *team*, returns the integer value of the corresponding action.

4.3.3 Evaluation of Generated Inference Code

For the experimental evaluation of the code generation scheme, 10 trainings with different seeds for the pseudo-random evolution process were run for each of the 5 selected games from the ALE [7]. Each of the 50 TPG graphs was trained for 400 generations using the meta-parameters described in [46]. These TPGs cover a wide range of use cases, with the smallest TPG consisting of only 3 *teams* and 8 *programs*, and the largest consisting of

25 *teams* and 50 *programs*. The performance of the 50 pre-trained TPGs was evaluated on 4 different platforms: an *Intel* Xeon CPU, an *Intel* i7 laptop CPU, an *Arm* A57 CPU from *NVidia's* Jetson Nano Tx2 platform, and an *Arm* A7 CPU from a Raspberry Pi2.

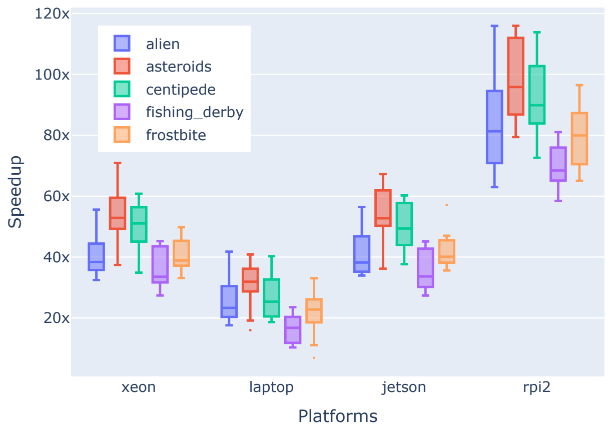


Figure 4.7: Speedup in inference time of the generated code with respect to the library.

The speedups in the inference time of the generated switch-based code compared to the library are shown in Figure 4.7. They are based on the total TPG inference time during a game and are presented as a ratio between the library and the generated code. Each box plot represents the statistics for the 10 TPGs trained for a given game and inferred on a given platform. On average across all games, the observed speedups are 44× on `xeon`, 24× on `laptop`, 45× on `jetson`, and 85× on `rpi2`. There are many possible reasons for this difference in average speedup between platforms, notably: different hardware complexity (in-order vs. out-of-order pipeline, bitwidth, instruction & data cache sizes), or different compiler versions. Nevertheless, the results obtained are very good, especially for the `rpi2`, which is the lightest CPU and benefits the most from the acceleration brought by code generation.

4.4 Additional Contributions on TPGs

In addition the contributions reviewed in Sections 4.2 and 4.3, other works using TPG were the object of personal contributions.

A first notable contribution is the work published in [C46], which investigates the ability of TPGs to classify data from a highly imbalanced data set. In general, when training an AI to classify data, the data-set presented during the training process is balanced so that the probability of observing a sample from any class is roughly equal. In an unbalanced data set, some classes have many more occurrences than others in the training data set. An unbalanced dataset introduces a bias in the training process [110], since predicting membership in a less frequent class is more *risky* for the trained AI agent, which will tend to avoid it. The work presented in [C46] shows that with an appropriate training loss function, TPGs exhibit great resilience to imbalanced data sets. The results are presented for data sets where a rare class has up to 10^4 less samples than a majority class.

Two other contributions present the use of TPGs in cybersecurity applications. In [C12], TPGs are used to identify wireless communicating devices by Radio Frequency (RF) fingerprint. Due to variations in the manufacturing process of wireless devices, each RF emission chain has its own small imperfections, called its fingerprint, which can be used to identify that device. Experimental results on an open database show that the performance of TPGs on this task is comparable to that of deep neural networks, for a fraction of their training cost.

In [J14], TPGs are used as a network intrusion detection system, which tries to detect malicious events by observing data packets passing through a computer network. The system built with TPGs is shown to be 8 times more energy efficient than random forests, a concurrent lightweight machine technique. On a state-of-the-art database, the proposed system is able to process 13.2k connections per second at a peak power consumption of less than 3.3 Watts.

4.5 Summary

This chapter introduced works on Tangled Program Graphs (TPGs), an innovative technique for building ultra-lightweight RL agents. The presented contributions aim at accelerating the training and inference of TPGs, both on General Purpose Processorss (GPPs) and on embedded hardware. All presented contributions are integrated into the open source GEGELATI framework, which was created and is being maintained by the VAADER team of the IETR.

The presented contributions were made possible thanks to the hard work and dedication of the following people:

- **PhD students**: Nicolas SOURBIER, Alice CHILLET
- **Master Students**: Pierre-Yves LE ROLLAND - RAUMER, Thomas BOURGOIN.

WALL.E: WALL-E.
EVE: WALL-E?
EVE: EVE.

Wall-E, N.62675

Chapter 5

Research Perspectives

Personal contributions of the last decade have been presented in previous chapters. These contributions share a common focus on studying various aspects of automatic optimization for embedded system design. This challenge is studied under the prism of model-based programming in Chapter 2, by integrating and characterizing approximations in Chapter 3, and for the design of lightweight Reinforcement Learning (RL) agents in Chapter 4.

This chapter formulates the research perspectives and challenges of personal interest for the foreseeable future of embedded system design. This chapter covers both short and long term perspectives, some of which are already funded and will be pursued over the next 3 years, while other more speculative ideas are still in an embryonic stage. Following the organization of this manuscript, the proposed perspectives are grouped into three main sections. Section 5.1 presents perspectives for model-based design automation of embedded systems. Section 5.2 proposes research goals for approximate computing and its use in the Video Coding for Machines (VCM) context. Then, Section 5.3 outlines research directions for the co-design of ultra-lightweight embedded AI. Finally, Section 5.4 concludes this chapter and manuscript.

5.1 Perspectives in Model-Based High-Performance Embedded System Programming

This section presents three sets of research perspectives: Section 5.1.1 explores the use of dataflow MoCs in the context of radio astronomy applications, Section 5.1.2 proposes new goals exploiting the idea of moldable parameters for the co-design of embedded systems, and Section 5.1.3 suggests potential long-term developments for dataflow-based CAD.

5.1.1 Dataflow Programming for Radio Astronomy HPC

Modern radio astronomy relies on large-scale scientific facilities such as the SKA radio telescope, whose hundreds of dishes and hundreds of thousands of antennas are spread over hundreds of thousands of square kilometers in South Africa and Australia [12]. These gigantic installations generate an astronomical amount of data, up to 50Tb per second, for which real-time processing is essential. The massive processing of this data relies on infrastructures that transform the electromagnetic waves picked up by the antennas into multispectral images of the sky. Optimizing the computations for such installations has many similarities and links to the field of embedded systems. Although at first glance part of the HPC domain, optimizing the use of computational resources is subject to strong real-time constraints and requires the use of many hardware elements typical of embedded systems, such as Field Programmable Gate Arrays (FPGAs) or GPUs. Moreover, driven by the fight against global warming, sustainability and frugality goals are becoming unavoidable for building HPC systems, imposing resource and power efficiency constraints similar to what is already the norm in embedded system design.

Since astronomers responsible for writing sky imaging algorithms are not computer scientists, they need to be provided with development tools that facilitate the deployment of these algorithms on complex targets. This is the goal of two ongoing research projects, the Dark ERA [27] and Rising STARS projects, which in particular have funded the work on application granularity control presented in Section 2.3.2.

The main research challenge in this context is to interface dataflow-based programming methods with HPC-oriented APIs and associated MoCs. While there are many software synthesis strategies for translating dataflow specifications into multicore code, as presented in Chapter 2, the unrelying imperative multi-threaded execution model is quite different from those used for programming High-Performance Computing (HPC) systems. In particular, the programming of multiple computing nodes interconnected by a high-speed local area network is typically accomplished using the communication-oriented Message Passing Interface (MPI) API. The use of GPU to accelerate computation relies on Single Instruction, Multiple Threads (SIMT) languages such as CUDA or OpenCL, which are closely related to the Bulk Synchronous Processing (BSP) MoC [38], and the programming of FPGAs mostly relies on the synchronous discrete event MoC.

A promising direction for programming HPC targets from dataflow MoCs is to automate the synthesis of high-level HPC-oriented API, such as MPI, CUDA, StarPU [4] or OpenMP [111]. Automatic code synthesis using these high-level APIs will allow to benefit from years of HPC research, while bringing the advantages of model-based programming to the built design flow. Although some preliminary work has been done on this topic, such as transforming dataflow graphs to make them more data-parallel and thus more GPU-friendly [44], or generating FPGA implementations [37], the full automation of the design flow is still far from tangible.

These goals will be pursued in particular in 2025 during a 6-month personal mobility to Swinburne University in Melbourne, Australia, funded by the Rising STARS project. The host team at Swinburne University specializes in the design and implementation of radio astronomy algorithms on HPC facilities.

5.1.2 Multi-Objective Co-Design Space Exploration with Moldable Dataflow MoCs

The papers presented in Section 2.4.2 introduced the concept of moldable parameters to the MoCs dataflow. In a nutshell, a moldable parameter in the application description is a parameter that affects the behavior of the application, functionally or non-functionally, but whose value is left undefined by the developer and is automatically set by the DSE framework. The seminal work [C28] on this topic only superficially explores the potential of this idea, and a deeper study represents an important perspective for future work.

On the software side, a key research goal is to propose intelligent heuristic algorithms for exploring the vast design space opened by moldable parameters. While it is shown in [C28] that the impact of most moldable parameters on optimized metrics can be classified, this result remains to be exploited for building a fast automated DSE. In a multi-objective context, where several potentially antagonistic metrics are optimized, an intelligent heuristic algorithm should automatically generate a front of Pareto-optimal configurations for the designer, while exploring a minimal subset of the design space.

Another direction for future work is to extend the idea of moldable parameters to the hardware side of the co-design flow. A moldable description of the target architecture could expose many settings to the DSE engine, such as variable numbers and types of PEs, different communication bus widths and speeds, different memory hierarchies, or different operating system options, such as real-time or not. As for the software counterpart, such a moldable architecture model would support multi-objective DSE for hardware-generative design flows. In the long term, building on such moldable models for architectures and applications will lay the foundation for building a fully automated co-design flow.

5.1.3 Long-Term Perspectives and Challenges in Model-Based Programming

Model-based design for programming embedded systems is still a very active research topic, with many promising directions for long-term future work, including

- **Using reconfigurable dataflow MoCs to support the execution of dynamic neural networks.** Dynamic neural networks introduce some dynamic control into the otherwise highly static dataflow of neural networks [34]. This dynamism can be used at a coarse granularity to disable processing of entire layers or channels, but also at a finer granularity, for example by dynamically adjusting the weights of the kernels. While these dynamic neural network models show promising results in terms of accuracy and theoretical complexity, their dynamism often makes them ill-suited for the GPU-oriented optimizations of most frameworks [34]. In fact, the execution model of GPUs mostly supports symmetric data-parallel computations, where a kernel is repeated many times without too much dynamism or asymmetry in the data path. Using reconfigurable dataflow MoCs, such as PISDF, is a promising direction to model and support efficient execution of these models on embedded hardware.

- **Bridging the gap between dataflow-based design and compilation with MLIR.** Most work on dataflow MoCs focuses on software or hardware synthesis from high-level semantics. This approach is inherently limited when it comes to jointly optimizing what happens at the graph level and what happens inside the actors, which are often viewed as black-box functions, with a few notable exceptions such as RVC-CAL [8]. Recently, Multi-Level Intermediate Representation (MLIR) has gained popularity as an intermediate layer between the compiler world and higher levels of abstraction [55]. A promising direction for dataflow research [16] is to study how the joint optimization of higher and lower levels of abstraction using the MLIR infrastructure could help produce more efficient software.

- **Proposal for a unified dataflow MoC semantics supporting data-parallel static, reconfigurable, and dynamic descriptions.** Although this problem has been studied many times, from Ptolemy's origin [58], to more recent attempts [92, 104], no solution is massively adopted in popular development environments or programming languages. On the contrary, most programming frameworks using dataflow-like MoCs continue to use the simplest static DAG-based models [2, 4, 83]. Therefore, a compelling MoC that combines the advantages of dynamic, reconfigurable, and static dataflow models with multidimensional capabilities [45] and a user-friendly API remains to be invented.

5.2 Perspectives in Approximate Computing

This section presents perspectives in the field of approximate computing, including domain-agnostic contributions in Section 5.2.1 and perspectives for the VCM domain in Section 5.2.2.

5.2.1 Source-to-Source Optimization of Data Types and SIMD Instructions

In the short term, work on approximate computing will continue during Baptiste Daniel Lamazière's PhD, in partnership with *WeDoLow*. *WeDoLow* is a startup founded by former PhD student Justine Bonnot, specialized in code optimization based on approximate computing techniques.

The first goal of this thesis is to automate the joint optimization of variable word length [P1] and the use of Single Instruction, Multiple Data (SIMD) instructions within an algorithm [71]. The novelty of this work is to consider the two intertwined problems as one, where the best global solution is most likely not a simple combination of the solutions of the two separate problems. In this context, the metrics to be optimized will be the latency of the system, as well as its memory footprint, its quality of service, and its energy footprint. The work will be developed as a transpiler tool, where the outputs of the optimization process are code change suggestions to the developer, supported by associated gains on the studied metrics.

A second goal of this thesis will be to combine the optimization of data word length and SIMD instructions with iterative compilation techniques [50]. Iterative compilation aims at automatically finding the best combination of compiler flags for a given code. In the context of this thesis, the goal is to use high-level knowledge of the compiled algorithm to constrain the design space explored during the iterative compilation process.

5.2.2 Global Optimization of VCM Systems

Alban Marie's PhD thesis showed that specializing a deep learning network for a specific type and intensity of compression noise makes it possible to achieve high resilience to compression artifacts [C37]. However, in scenarios where multiple compression algorithms feed images into a single AI, it becomes quite impractical to store a specific network for each type of noise. A possible direction for future work would be to study how a network is affected when it is tuned for a specific type of noise. The goal of this study would be to isolate the minimal parts of the network that require specialization in order to limit the effects of backpropagation to those parts when specializing a network. Thus, instead of storing a whole dedicated network for each type of compression noise, only a minimal subset of weights of the network could be swapped to have a lightweight specialization of the network for different types of compression noise. This approach could be combined with neural network quantization techniques [30] to further minimize the memory required for specialized weights.

Another perspective in the continuation of Alban MARIE's thesis is the integration of VCM-oriented quality metrics into image or video encoders. As shown in [C36], common image quality metrics such as SSIM or PSNR do not correlate well with the quality of an image as perceived by an AI. This lack of correlation is due to the assumption that higher quality measured by a metric should lead to better accuracy of the AI algorithm, which is not the case for these reference metrics. Since these reference metrics correlate well with the subjective quality perceived by human psychovisual systems, they are used at the core of most rate-distortion optimization processes of image encoders. The proposed direction for future work is therefore to develop new image quality metrics that correlate well with the fitness of AIs, and to integrate them as part of the rate-distortion optimization process of VCM-oriented encoders.

As a first long-term perspective in this area, automating the global optimization of a VCM pipeline is the most challenging. Previous contributions have mostly focused on a specific part of a VCM system, either on the AI side [C37] or on the encoder side [C35, C36, C38]. Building on these studies, a global optimization of the whole VCM system could be done, where all stages of the pipeline are optimized together. For example, from a global perspective, optimizing the energy of the pre-processing, the encoder, the RF transmission, and the receiving AI may yield completely different solutions than optimizing each part separately, where more energy may be spent in one part to help save some energy elsewhere.

A final long-term perspective in the field of lossy coding for machines is to extend the work done on images and videos to other stream processing application domains. Indeed, it would be interesting to study the adaptation of the results from the VCM domain to the processing of audio, radar, or biosignals.

5.3 Perspectives in Ultra-Lightweight AI for Embedded Systems

5.3.1 Full-Stack Optimization for Ultra-Low Power AIs

During the next three years, the research on TPGs will be funded by the FOUTICS (Full-stack Optimization of Ultra-low-power TPGs for Intelligent Cyberphysical Systems) project, co-led with Mickaël DARDAILLON. The general objective of FOUTICS is to propose a full-stack machine learning technique, with multidisciplinary contributions to the TPG model, training and inference implementation, to achieve new performance at ultra-low power.

The scientific goals of the FOUTICS project are threefold:

- **Extend the TPG learning capabilities:** In addition to improving the efficiency of TPG on existing environments, model extensions will open up new types of learning environments, such as continuous action spaces or non-reinforcement learning environments, needed to extend TPG to cyber-physical system use cases.

- **Integrate energy optimizations at the core of the TPG training process:** Overall energy optimization will minimize the energy consumption of the computing system for TPG training and inference, as well as, where relevant, the energy consumption of the physical actuators of the controlled RL environment.

- **Propose highly efficient implementation techniques:** In order to find the most suitable hardware platform for TPGs, the implementation will be pursued and compared on several state-of-the-art hardware. The implementation will be investigated for both efficient training and inference on battery-powered ultra-low-power devices and reconfigurable devices for nanosecond response times.

Beyond the FOUTICS goals, a potential direction for future work is to combine several bio-inspired AI techniques to study their competitiveness with deep learning. For example, combining spiking neurons [31] with TPGs could be an interesting research direction to build ultra-lightweight AI.

5.3.2 Perspectives of Design Automation for Frugal and Embedded AI

The pervasive use of AI and the urgent need for more sustainable systems make the search for frugal AI an inescapable goal for the next decades, as a research perspective for embedded system design automation. While many techniques exist to reduce the complexity of deep learning AIs, such as pruning, quantization, or compression, their impact on optimized network complexity and accuracy is still difficult to predict. An interesting direction for future work, explored in Mewe KAHANAM's thesis, is to build analytical models to predict the impact of such optimization techniques on a network for a given hardware. A key advantage of such analytical methods is that they could be performed *a priori*, that is, without actually performing the optimizations and associated re-training that is often required to measure their efficiency. Such a prediction technique would be a key asset when exploring the design optimization space of an AI-based system, and could help select the appropriate hardware and network architecture without relying on the best guesses of developers or a brute-force approach.

5.4 Conclusion

This habilitation manuscript presents personal research works developed over the last 10 years, in the area of embedded system design automation. Along the 4 main chapters, the contributions cover a wide but coherent variety of topics, ranging from model-based programming to evolution-inspired AI algorithms to approximate computing and VCM. As this manuscript attests, the life of a researcher is full of surprises, and these contributions far outshine the perspectives established at the end of my PhD work, or in the research project presented for my hiring as an associate professor. Thus, these final chapters have presented only a tiny fraction of the research perspectives opened up by the research to date, which will most likely continue in new, unpredictable, and exciting directions.

Glossary

AI Artificial Intelligence. ix, 5, 13, 15, 17, 18, 43, 44, 54, 55, 56, 58, 60, 61, 62, 70, 73, 74, 75, 79, 80, 81, 82

ALE Arcade Learning Environment. 65, 69, 72

ALU Arithmetic and Logic Unit. 46

API Application Programming Interface. 2, 76, 79

ATER Attaché Temporaire d'Enseignement et de Recherche. 1

BDR Bjøntegaard-Delta Rate. 59

BSP Bulk Synchronous Processing. 76

Cassis Characterization with Adaptive Sample-Size Inferential Statistics. 50, 51, 52

CAD Computer Aided Design. vii, 9, 12, 16, 24, 26, 32, 75, 87

CNN Convolutional Neural Network. 54, 56, 57, 87

CPU Core Processing Unit. 5, 38, 45, 50, 69, 73, 88

DAG Directed Acyclic Graph. 26, 27, 34, 35, 36, 37, 78, 87

DNN Deep Neural Network. 46, 61

DSE Design Space Exploration. vii, 4, 19, 25, 26, 32, 36, 37, 38, 39, 40, 41, 42, 47, 48, 52, 53, 54, 60, 77

DSP Digital Signal Processing. 5, 21

DVFS Dynamic Voltage and Frequency Scaling. 10

Fifo First-in, First-Out queue. 20, 21, 22, 23, 25, 29, 31, 37, 39

Foutics Full-stack Optimization of Ultra-low-power TPGs for Intelligent Cyberphysical Systems. 81

FIR Finite Impulse Response. 54

FPGA Field Programmable Gate Array. 76, 77

FSM Finite State Machine. 72

Gegelati Generic Evolvable Graphs for Efficient Learning of Artificial Tangled Intelligence. 66, 68, 69, 71, 74

GPP General Purpose Processors. 3, 5, 10, 12, 74

GPU Graphics Processing Unit. 6, 10, 12, 61, 66, 76, 77, 78

HDR Habilitation à Diriger des Recherches. 1

HEVC High-Efficiency Video Coding. 54

HLS High-Level Synthesis. 6

HPC High-Performance Computing. ix, 13, 16, 42, 66, 67, 76, 77

IETR Institut d'Electronique et des Technologies du numéRique. 1, 66, 74

IoT Internet-of-Things. 6, 54, 55, 56, 61

ISA Instruction Set Architecture. 5

LOA Lower-part Or Adder. 49, 51, 87

MEG Memory Exclusion Graph. 37, 38

MLIR Multi-Level Intermediate Representation. 78

MoA Model of Architecture. 4, 24

MoC Model of Computation. vii, ix, 3, 8, 11, 14, 16, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 32, 33, 34, 36, 37, 38, 39, 40, 41, 42, 75, 76, 77, 78

MPI Message Passing Interface. 76

MPSoC Multiprocessor System-on-Chip. 5, 8, 9, 10, 12, 14, 17, 19, 25, 33, 38, 42, 67

NLP Natural Language Processing. 61

NUMA Non-Uniform Memory Access. 38

PE Processing Element. 5, 33, 38, 77

PISDF Parameterized and Interfaced SDF. 8, 22, 23, 24, 25, 27, 30, 31, 32, 33, 35, 36, 38, 39, 78, 87

PRNG Pseudo-Random Number Generator. 67, 68, 69

QoS Quality of Service. 46, 47, 48

RF Radio Frequency. 15, 74, 80

RISC-V Reduced Instruction Set Computer (RISC) five. 4, 5

RL Reinforcement Learning. 15, 62, 63, 70, 74, 75, 81, 87

Spider Synchronous Parameterized and Interfaced Dataflow Embedded Runtime. 24, 38

SAD State-Aware Dataflow. 29, 30, 31, 87

SCAPE Scaling up of Clusters of Actors on Processing Element. 33, 34, 35, 36, 37, 87

SDF Synchronous Dataflow. 8, 21, 22, 23, 25, 26, 27, 28, 30, 31, 32, 33, 34, 37, 39, 85, 87

SIMD Single Instruction, Multiple Data. ix, 79

SIMT Single Instruction, Multiple Threads. 76

SKA Square Kilometer Array. 16, 76

SRV Single Repetition Vector. 35

TPG Tangled Program Graph. viii, 16, 17, 18, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 81, 87, 88

TPU Tensor Processing Unit. 6

URC Unique Repetition Count. 35

Vaader Video Analysis and Architecture Design for Embedded Resources. 1, 74

VCM Video Coding for Machines. ix, 15, 16, 17, 43, 48, 54, 55, 56, 57, 58, 60, 75, 79, 80, 82, 87

WCET Worst-Case Execution Time. 27

List of Figures

List of Tables

Personal Publications

Book chapters & Thesis

[B1] J. Bonnot, D. Menard, and K. Desnos. “Analysis of the Impact of Approximate Computing on the Application Quality”. In: *Approximate Computing Techniques: From Component- to Application-Level*. Ed. by A. Bosio, D. Menard, and O. Sentieys. Springer, June 2022. Chap. 6, pp. 145–176. published.

[B2] J. Castrillon, K. Desnos, A. Goens, and C. Menard. “Dataflow Models of Computation for Programming Heterogeneous Multicores”. In: *Handbook of Computer Architecture*. Springer, May 2021. Forthcoming (cit. on pp. 11, 19).

[B3] K. Desnos and F. Palumbo. “Dataflow Modeling for Reconfigurable Signal Processing Systems”. In: *Handbook of Signal Processing Systems*. Ed. by S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala. Third. Springer, Jan. 2018. published (cit. on pp. 11, 12, 19, 38).

[B4] K. Desnos. “Memory Study and Dataflow Representations for Rapid Prototyping of Signal Processing Applications on MPSoCs”. Thèse de doctorat dirigée par Nezan, Jean-François, Rennes, INSA 2014. PhD thesis. 2014. URL: http://www.theses.fr/2014ISAR0004 (cit. on pp. 8, 26, 37).

Journal articles

[J1] M. Ammar, M. Baklouti, M. Pelcat, K. Desnos, and M. Abid. “Comparing Three Clustering-based Scheduling Methods for Energy-Aware Rapid Design of MP2SoCs”. In: *Journal of Signal Processing Systems (JSPS)* (Aug. 2017). Ed. by Springer. published.

[J2] M. Ammar, M. Baklouti, M. Pelcat, K. Desnos, and M. Abid. “MDE-based Rapid DSE of multi-core embedded systems: The H.264 Decoder Case Study”. In: *Informacije Midem-journal of Microelectronics Electronic Components and Materials* 46.4 (Jan. 2016), pp. 219–228. URL: https://hal-univ-rennes1.archives-ouvertes.fr/hal-01502348. published.

[J3] F. Arrestier, K. Desnos, E. Juarez, and D. Menard. “Numerical Representation of Directed Acyclic Graphs for Efficient Dataflow Embedded Resource Allocation”. In: *Transactions on Embedded Computing Systems - EMSOFT Proceedings* 18.55 (Oct. 13, 2019), pp. 1–22. published (cit. on pp. 11, 14, 16, 19, 27).

[J4] J. Bonnot, V. Camus, K. Desnos, and D. Menard. “Adaptive Simulation-based Framework for Error Characterization of Inexact Circuits”. In: *Microelectronics Reliability* (Feb. 2019). Ed. by Elsevier. published.

[J5] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi. “Memory Analysis and Optimized Allocation of Dataflow Applications on Shared-Memory MPSoCs”. In: *Journal of Signal Processing Systems* 80.1 (Nov. 1, 2014). Ed. by Springer, pp. 19–37. URL: https://doi.org/10.1007/s11265-014-0952-6. published (cit. on pp. 8, 10, 13, 36, 37).

[J6] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi. “On Memory Reuse Between Inputs and Outputs of Dataflow Actors”. In: *Transactions on Embedded Computing Systems (TECS)* 15.2 (2016). Ed. by ACM, 30:1–30:25. URL: http://doi.acm.org/10.1145/2871744. published (cit. on pp. 8, 10, 11).

[J7] A. Enrici, J. Lallet, R. Pacalet, L. Apvrille, K. Desnos, and I. Latif. “Model-Based Programming for Multi-processor Platforms with TTool/DIPLODOCUS and OMC”. In: *Communications in Computer and Information Science (Extended and revised paper from Model-Driven Engineering and Software Development (MODELSWARD))* (Jan. 2019). Ed. by Springer, pp. 56–81. published.

[J8] R. Lazcano, D. Madroñal, R. Salvador, K. Desnos, M. Pelcat, R. Guerra, H. Fabelo, S. Ortega, S. Lopez, G. M. Callico, E. Juarez, and C. Sanz. “Porting a PCA-based hyperspectral image dimensionality reduction algorithm for brain cancer detection on a manycore architecture”. In: *Journal of Systems Architecture (JSA)* 77 (Oct. 2017). Ed. by Elsevier, pp. 101–111. URL: http://www.sciencedirect.com/science/article/pii/S1383762116302934. published (cit. on p. 13).

[J9] Y. Lee, Y. Liu, K. Desnos, L. Barford, and S. S. Bhattacharyya. “Passive-Active Flowgraphs for Efficient Modeling and Design of Signal Processing Systems”. In: *Journal of Signal Processing Systems* 92.10 (July 2020), pp. 1133–1151. published (cit. on pp. 11, 19, 28).

[J10] D. Madroñal, F. Arrestier, J. Sancho, A. Morvan, R. Lazcano, K. Desnos, R. Salvador, D. Menard, E. Juarez, and C. Sanz. “PAPIFY: automatic instrumentation and monitoring of dynamic dataflow applications based on PAPI”. In: *IEEE Access* (Aug. 2019). published (cit. on p. 16).

[J11] A. Mercat, J. Bonnot, M. Pelcat, K. Desnos, W. Hamidouche, and D. Menard. “Smart search space reduction for approximate computing: A low energy HEVC encoder case study”. In: *Journal of Systems Architecture (JSA)* 80 (Oct. 2017). Ed. by Elsevier, pp. 56–67. published (cit. on p. 13).

[J12] M. Pelcat, A. Mercat, K. Desnos, L. Maggiani, Y. Liu, J. Heulot, J. Nezan, W. Hamidouche, D. Ménard, and S. S. Bhattacharyya. “Reproducible Evaluation of System Efficiency With a Model of Architecture: From Theory to Practice”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.10 (Oct. 2018), pp. 2050–2063 (cit. on pp. 4, 24).

[J13] C. Rubattu, F. Palumbo, C. Sau, R. Salvador, J. Sérot, K. Desnos, L. Raffo, and M. Pelcat. “Dataflow-Functional High-Level Synthesis for Coarse-Grained Reconfigurable Accelerators”. In: *Embedded Systems Letters* (Nov. 2018). Ed. by IEEE, p. 4. published.

[J14] N. Sourbier, K. Desnos, T. Guyet, F. Majorczyk, O. Gesny, and M. Pelcat. “SECURE-GEGELATI Always-On Intrusion Detection through GEGELATI Lightweight Tangled Program Graphs”. In: *Journal of Signal Processing Systems* (Nov. 2021). Forthcoming (cit. on pp. 13, 62, 74).

[J15] L. Suriano, F. Arrestier, A. Rodriguez, K. Desnos, J. Heulot, M. Pelcat, and E. de la Torre. “DAMHSE: Programming Heterogeneous MPSoCs with Hardware Acceleration using Dataflow-based Design Space Exploration and Automated Rapid Prototyping”. In: *Microprocessors and Microsystems: Embedded Hardware Design (MICPRO)* 71 (Sept. 2019). published (cit. on pp. 10, 16).

[J16] Z. Zhou, W. Plishker, S. S. Bhattacharyya, K. Desnos, M. Pelcat, and J.-F. Nezan. “Scheduling of Parallelized Synchronous Dataflow Actors for Multicore Signal Processing”. In: *Journal of Signal Processing Systems (JSPS)* 83.3 (Oct. 2014). Ed. by Springer, pp. 309–328. URL: https://doi.org/10.1007/s11265-014-0956-2. published.

Patent

[P1] J. Bonnot, D. Ménard, and K. Desnos. *Procédé et dispositif d’optimisation de longueurs de représentation de variables*. Apr. 2019 (cit. on pp. 15, 44, 53, 79).

Conference articles

[C1] M. Ammar, M. Baklouti, M. Pelcat, K. Desnos, and A. Mohamed. “Automatic Generation of S-LAM Descriptions from UML/MARTE for the DSE of Massively Parallel Embedded Systems”. In: *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. Takamatsu, Japan.: Springer International Publishing, Jan. 2016, pp. 195–211. URL: https://hal.archives-ouvertes.fr/hal-01252511/file/ammar_automatic_2015.pdf. published.

[C2] M. Ammar, M. Baklouti, M. Pelcat, K. Desnos, and A. Mohamed. "MARTE to PiSDF transformation for data-intensive applications analysis". In: *Conference on Design and Architectures for Signal and Image Processing (DASIP)*. Madrid, Spain, Oct. 2014. URL: https://hal.archives-ouvertes.fr/hal-01122725/file/DASIP_2014_submission_27.pdf. published.

[C3] M. Ammar, M. Baklouti, M. Pelcat, K. Desnos, and A. Mohamed. "Off-Line DVFS Integration in MDE-Based Design Space Exploration Framework for MP2SoC Systems". In: *International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. Ed. by IEEE. Paris, France, June 2016, pp. 160–165. published.

[C4] M. Ammar, M. Baklouti, M. Pelcat, K. Desnos, and A. Mohamed. "On Exploiting Energy-Aware Scheduling Algorithms for MDE-Based Design Space Exploration of MP2SoC". In: *International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. Heraklion, Greece, Feb. 2016, pp. 643–650. URL: https://hal.archives-ouvertes.fr/hal-01305971/file/ammar_exploiting_2016.pdf. published.

[C5] F. Arrestier, K. Desnos, M. Pelcat, J. Heulot, E. Juarez, and D. Menard. "Delays and States in Dataflow Models of Computation". In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. Pythagorion, Greece: ACM, July 15, 2018, pp. 47–54. published (cit. on pp. 14, 16, 29–31).

[C6] J. Bonnot, V. Camus, K. Desnos, and D. Menard. "CASSIS: Characterization with Adaptive Sample-Size Inferential Statistics Applied to Inexact Circuits". In: *European Signal Processing Conference (EUSIPCO)*. Roma, Italy: IEEE, Sept. 3, 2018, pp. 677–681. published (cit. on pp. 15, 44, 46, 50, 52).

[C7] J. Bonnot, K. Desnos, and D. Menard. "Accuracy Evaluation Based on Simulation for Finite Precition Systems Using Inferential Statistics". In: *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Brighton, UK, May 2019. published (cit. on pp. 9, 48).

[C8] J. Bonnot, K. Desnos, and D. Menard. "Algorithm-Level Approximation for Fast (or not) Embedded Stereovision Algorithm". In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. Pythagorion, Greece: ACM, July 2018. URL: https://hal.archives-ouvertes.fr/hal-01879595. published (cit. on pp. 9, 13, 46, 52).

[C9] J. Bonnot, K. Desnos, and D. Menard. "Fast Kriging-based Error Evaluation for Approximate Computing Systems". In: *Design, Automation and Test in Europe Conference (DATE)*. ACM, Mar. 9, 2020. published (cit. on pp. 12, 15, 44, 48, 53, 54).

[C10] J. Bonnot, K. Desnos, and D. Menard. “Stochastic Modeling to Accelerate Approximate Operators Simulation”. In: *International Symposium on Circuits and Systems (ISCAS)*. May 2018. URL: https://hal.archives-ouvertes.fr/hal-01812706/document. published.

[C11] J. Bonnot, K. Desnos, M. Pelcat, and D. Menard. “A Fast and Fuzzy Functional Simulator of Inexact Arithmetic Operators for Approximate Computing Systems”. In: *Great Lakes Symposium on VLSI (GLSVLSI)*. Chicago, USA: ACM, May 2018. URL: https://hal.archives-ouvertes.fr/hal-01812719/document. published (cit. on pp. 15, 46–48, 53).

[C12] A. Chillet, B. Boyer, R. Gerzaguet, K. Desnos, and M. Gautier. “Tangled Program Graph for Radio-Frequency Fingerprint Identification”. In: *International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*. Toronto, Canada: IEEE, Sept. 2023. published (cit. on pp. 62, 74).

[C13] A. Chillet, R. Gerzaguet, K. Desnos, M. Gautier, E. S. Lohan, E. Nogues, and M. Valkama. “How to Design a Channel-Resilient Database for Radio Frequency Fingerprint Identification?” In: *International Conference on Communications (ICC)*. Denver, CO, USA: IEEE, June 2024. published.

[C14] H. Deroui, K. Desnos, J.-F. Nezan, and A. Munier-Kordon. “Relaxed Subgraph Execution Model for the Throughput Evaluation of IBSDF Graphs”. In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. Samos, Greece, Jan. 1, 2017. URL: https://hal.archives-ouvertes.fr/hal-01569593/file/32_Final_Paper.pdf. published (cit. on pp. 14, 19, 23, 25, 32).

[C15] H. Deroui, K. Desnos, J.-F. Nezan, and A. Munier-Kordon. “Throughput Evaluation of DSP Applications based on Hierarchical Dataflow Models”. In: *International Symposium on Circuits and Systems (ISCAS)*. Baltimore, MD, USA, Jan. 1, 2017. URL: https://hal.archives-ouvertes.fr/hal-01514641/file/ISCAS2017%20_published.pdf. published (cit. on pp. 12, 14, 19, 23, 25, 32).

[C16] K. Desnos, S. E. Assad, A. Arlicot, M. Pelcat, and D. Menard. “Efficient multicore implementation of an advanced generator of discrete chaotic sequences”. In: *International Workshop on Chaos-Information Hiding and Security (C-IHS)*. London, UK, Dec. 2014, pp. 31–36. URL: https://hal.archives-ouvertes.fr/hal-01094677/file/ICITST2014-ID-103_v35.pdf. published.

[C17] K. Desnos, T. Bourgoin, N. Sourbier, M. Dardaillon, O. Gesny, and M. Pelcat. “Ultra-Fast Machine Learning Inference through C Code Generation for Tangled Program Graphs”. In: *International Workshop on Signal Processing Systems (SiPS)*. Rennes, France: IEEE, Nov. 2, 2022. published (cit. on pp. 11, 12, 62, 71).

[C18] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi. "Buffer merging technique for minimizing memory footprints of Synchronous Dataflow specifications". In: *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Brisbane, Australia, Apr. 1, 2015, pp. 1111–1115. URL: https://hal.archives-ouvertes.fr/hal-01146340/file/20150423-ICASSP15_Desnos_perso%20%281%29.pdf. published (cit. on p. 8).

[C19] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi. "Distributed Memory Allocation Technique for Synchronous Dataflow Graphs". In: *International Workshop on Signal Processing Systems (SiPS)*. Dallas, TX, USA, Oct. 1, 2016, pp. 45–50. URL: https://hal.archives-ouvertes.fr/hal-01390486/document. published (cit. on pp. 8, 10, 11, 19, 38).

[C20] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi. "Memory bounds for the distributed execution of a hierarchical Synchronous Data-Flow graph". In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. Samos, Greece, July 1, 2012, pp. 160–167. URL: https://hal.archives-ouvertes.fr/hal-00721335/file/Samos12.pdf. published (cit. on pp. 8, 25, 37).

[C21] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi. "Pre- and post-scheduling memory allocation strategies on MPSoCs". In: *Electronic System Level Synthesis Conference (ESLsyn)*. Austin, TX, USA: IEEE, May 1, 2013, pp. 1–6. URL: https://hal.archives-ouvertes.fr/hal-00868945/file/ESLSyn13.pdf. published (cit. on pp. 8, 10, 11, 37).

[C22] K. Desnos, M. Pelcat, J.-F. Nezan, S. S. Bhattacharyya, and S. Aridhi. "PiMM: Parameterized and Interfaced dataflow Meta-Model for MPSoCs runtime reconfiguration". In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. Samos, Greece, July 1, 2013, pp. 41–48. URL: http://kdesnos.fr/wp-content/uploads/publis/Desnos_Pimm_2013.pdf. published (cit. on pp. 8, 22, 39).

[C23] K. Desnos, N. Sourbier, P.-Y. Raumer, O. Gesny, and M. Pelcat. "GEGELATI: Lightweight Artificial Intelligence through Generic and Evolvable Tangled Program Graphs". In: *Workshop on Design and Architectures for Signal and Image Processing (DASIP)*. International Conference Proceedings Series (ICPS). Budapest, Hungary: ACM, Jan. 18, 2021. published (cit. on pp. 10, 11, 13, 62, 66, 68).

[C24] G. Georgakarakos, S. Kanur, J. Lilius, and K. Desnos. "Task-based Execution of Synchronous Dataflow Graphs for Scalable Multicore Computing". In: *International Workshop on Signal Processing Systems (SiPS)*. Ed. by IEEE. Lorient, France, Jan. 2017. published.

[C25] J. Hascoët, K. Desnos, J.-F. Nezan, and B. D. de Dinechin. "Hierarchical Dataflow Model for efficient programming of clustered manycore processors". In: *International Conference on Application-specific Systems, Architectures and Pro-*

cessors (ASAP). Ed. by IEEE. Seattle, WA, USA, July 2017, pp. 137–142. URL: https://hal.archives-ouvertes.fr/hal-01564019/file/ASAP-2017-Hierarchical-Dataflow-Model.pdf. published (cit. on p. 14).

[C26] J. Hascoet, B. D. de Dinechin, K. Desnos, and J.-F. Nezan. "A Distributed Framework for Low-Latency OpenVX over the RDMA NoC of a Clustered Manycore". In: *High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, Sept. 2018, pp. 1–7. published (cit. on p. 10).

[C27] J. Heulot, M. Pelcat, K. Desnos, J.-F. Nezan, and S. Aridhi. "Spider: A Synchronous Parameterized and Interfaced Dataflow-based RTOS for multicore DSPS". In: *Embedded Design in Education and Research Conference (EDERC)*. Milan, Italy, Sept. 1, 2014, pp. 167–171. URL: https://hal.archives-ouvertes.fr/hal-01067052/file/ederc2014.pdf. published (cit. on pp. 8, 10, 11, 16, 24, 26, 38).

[C28] A. Honorat, T. Bourgoin, H. Miomandre, K. Desnos, D. Menard, and J.-F. Nezan. "Influence of Dataflow Graph Moldable Parameters on Optimization Criteria". In: *Workshop on Design and Architectures for Signal and Image Processing (DASIP)*. Budapest, Hungary, June 20, 2022. published (cit. on pp. 9, 10, 12, 14, 19, 39–41, 48, 77).

[C29] A. Honorat, K. Desnos, S. S. Bhattacharyya, and J.-F. Nezan. "Scheduling of Synchronous Dataflow Graphs with Partially Periodic Real-Time Constraints". In: *Conference on Real-Time Networks and Systems (RTNS)*. ICPS. Paris, France: ACM, June 2020, pp. 22–33. published (cit. on pp. 10, 11, 13, 14, 19, 25, 27).

[C30] A. Honorat, K. Desnos, M. Dardaillon, and J.-F. Nezan. "A Fast Heuristic to Pipeline SDF Graphs". In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. Vol. 12471. LNCS. Pythagorion, Greece: Springer, July 2020. published (cit. on pp. 12, 14, 19, 33).

[C31] A. Honorat, K. Desnos, M. Pelcat, and J.-F. Nezan. "Modeling Nested For Loops with Explicit Parallelism in Synchronous Dataflow Graphs". In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. Vol. 11733. LNCS. Pythagorion, Greece: Springer, July 2019, pp. 269–280. published (cit. on pp. 19, 28, 40).

[C32] R. Lazcano, D. Madronal, K. Desnos, M. Pelcat, R. Guerra, S. Lopez, E. Juarez, and C. Sanz. "Parallelism Exploitation of a Dimensionality Reduction Algorithm Applied to Hyperspectral Images". In: *Conference on Design and Architectures for Signal and Image Processing (DASIP)*. Rennes, France, Jan. 2016. URL: https://hal.archives-ouvertes.fr/hal-01415948. published.

[C33] R. Lazcano, I. Sidrach-Cardona, D. Madroñal, K. Desnos, M. Pelcat, E. Juarez, and C. Sanz. "Parallelism exploitation of a PCA algorithm for hyperspectral images using RVC-CAL". In: *SPIE Remote Sensing*. Edimburgh, UK, Jan. 2016. published.

[C34] D. Madroñal, R. Lazcano, A. Morvan, R. Salvador, K. Desnos, E. Juarez, and C. Sanz. “Automatic Instrumentation of Dataflow Application using PAPI”. In: *International Conference on Computing Frontiers.* Ischia, Italy: ACM, May 2018, pp. 232–235. published.

[C35] A. Marie, F. Chen, K. Desnos, J. Zhou, L. Morin, and L. Zhang. “Towards Machine Perception Aware Image Quality Assessment”. In: *25th International Workshop on Multimedia Signal Processing.* Poitiers, France: IEEE, Sept. 2023. published (cit. on p. 80).

[C36] A. Marie, K. Desnos, L. Morin, and L. Zhang. “Evaluation of Image Quality Assessment Metrics for Semantic Segmentation in a Machine-to-Machine Communication Scenario”. In: *International Conference on Quality of Multimedia Experience (QoMEX).* Ghent, Belgium: IEEE, June 2023. published (cit. on p. 80).

[C37] A. Marie, K. Desnos, L. Morin, and L. Zhang. “Expert Training: Enhancing AI Resilience to Image Coding Artifacts”. In: *Electronic Imaging, Image Processing: Algorithms and Systems XX.* Society for Imaging Science and Technology, Jan. 17, 2022. Forthcoming (cit. on pp. 13, 44, 57, 58, 79, 80).

[C38] A. Marie, K. Desnos, L. Morin, and L. Zhang. “Video Coding for Machines: Large-Scale Evaluation of DNNs Robustness to Compression Artifacts for Semantic Segmentation”. In: *International Workshop on Multimedia Signal Processing (MMSP).* Shanghai, China: IEEE, Sept. 26, 2022. published (cit. on pp. 12, 13, 44, 46, 57–60, 80).

[C39] H. Miomandre, J. Hascoët, K. Desnos, K. Martin, B. D. D. Dinechin, and J.-F. Nezan. “Embedded Runtime for Reconfigurable Dataflow Graphs on Manycore Architectures”. In: *PARMA-DITAM.* Manchester, United Kingdom, Jan. 2018. URL: `https://hal.archives-ouvertes.fr/hal-01704702`. published (cit. on pp. 11, 16, 19, 38).

[C40] F. Palumbo, T. Fanni, C. Sau, A. Rodriguez, D. Madroñal, K. Desnos, A. Morvan, M. Pelcat, C. Rubattu, R. Lazcano, L. Raffo, E. de la Torre, E. Juarez, C. Sanz, and P. S. de la Roja. “Hardware/Software Self-Adaptation in CPS: the CERBERO Project Approach”. In: *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS).* Vol. 11733. LNCS. Pythagorion, Greece: Springer, July 2019, pp. 416–428. published.

[C41] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi. “Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming”. In: *Embedded Design in Education and Research Conference (EDERC).* Milan, Italy, Sept. 2014, pp. 36–40. URL: `https://hal.archives-ouvertes.fr/hal-01059313/file/ederc2014.pdf`. published (cit. on pp. 16, 24, 26).

[C42] M. Pelcat, K. Desnos, L. Maggiani, Y. Liu, J. Heulot, J.-F. Nezan, and S. S. Bhattacharyya. “Models of Architecture: Reproducible Efficiency Evaluation for Signal Processing Systems”. In: *International Workshop on Signal Processing Systems (SiPS)*. Ed. by IEEE. Dallas, TX, USA, Oct. 2016, pp. 121–126. URL: https://hal.archives-ouvertes.fr/hal-01390508/file/sips_moa.pdf. published.

[C43] O. Renaud, D. Gageot, K. Desnos, and J.-F. Nezan. “SCAPE: HW-Aware Clustering of Dataflow Actors for Tunable Scheduling Complexity”. In: *Workshop on Design and Architectures for Signal and Image Processing (DASIP)*. Toulouse, France: Springer, Jan. 2023. published (cit. on pp. 9, 11, 12, 14, 19, 25, 26, 33, 37).

[C44] O. Renaud, N. Haggui, K. Desnos, and J.-F. Nezan. “Automated Clustering and Pipelining of Dataflow Actors for Controlled Scheduling Complexity”. In: *European Signal Processing Conference (EUSIPCO)*. Helsinki, Finland: IEEE, Sept. 2023. published.

[C45] H. Rexha, S. Lafond, and K. Desnos. “Energy-Efficient Actor Execution for SDF Application on Heterogeneous Architectures”. In: *International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. Mar. 2018. published (cit. on pp. 10, 11).

[C46] N. Sourbier, J. Bonnot, O. Gesny, F. Majorczyk, K. Desnos, T. Guyet, and M. Pelcat. “Imbalanced Classification with TPG Genetic Programming: Impact of Problem Imbalance and Selection Mechanisms”. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO)*. Boston, USA: ACM, July 2022. published (cit. on pp. 62, 73, 74).

[C47] L. Suriano, A. Rodriguez, K. Desnos, M. Pelcat, and E. de la Torre. “Analysis of a Heterogeneous Multi-Core, Multi-HW-Accelerator-Based System Designed Using PREESM and SDSoC”. In: *International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. Madrid, Spain, July 2017, pp. 1–7. published (cit. on p. 12).

[C48] S. Wang, N. Gac, H. Miomandre, J.-F. Nezan, K. Desnos, and F. Orieux. “Algorithmic and Hardware Design Space Exploration for Radio-Interferometric Algorithms”. In: *Workshop on Design and Architectures for Signal and Image Processing (DASIP)*. LNCS. Springer, Jan. 2024. published.

[C49] Z. Zhou, K. Desnos, M. Pelcat, J.-F. Nezan, W. Plishker, and S. S. Bhattacharyya. “Scheduling of parallelized synchronous dataflow actors”. In: *International Symposium on System on Chip (SoC)*. Tampere, Finland, Oct. 2013, pp. 1–10. published.

Technical Reports & Project Deliverables

[T1] T. Bourgoin, N. Sourbier, M. Dardaillon, and K. Desnos. *Génération de code pour une bibliothèque d'apprentissage par renforcement.* Tech. rep. INSA Rennes, Oct. 2021. URL: http://kdesnos.fr/wp-content/uploads/publis/20211004-Internship_Report_Bourgoin_final.pdf (cit. on p. 62).

[T2] M. Dejean-Servières, K. Desnos, K. Abdelouahab, W. Hamidouche, L. Morin, and M. Pelcat. *Study of the Impact of Standard Image Compression Techniques on Performance of Image Classification with a Convolutional Neural Network.* Intership report. INSA Rennes, Dec. 2017 (cit. on pp. 44, 46, 57).

[T3] K. Desnos. *GEGELATI: Reinforcement Learning Framework with Tangled Program Graphs.* Tech. rep. Feb. 2020. URL: https://www.youtube.com/watch?v=t0Ta5Vo5h7s (cit. on p. 62).

[T4] K. Desnos, F. Arrestier, A. Morvan, M. Pelcat, D. Madronal, and E. Juarez. *Energy Optimized Sensor-Based Adaptive Software on Heterogeneous Platforms.* Tech. rep. CERBERO H2020-ICT-2016, 2018. URL: https://youtu.be/a9WIucWfjkU (cit. on p. 38).

[T5] K. Desnos, M. Pelcat, J. Oliveira, C. Sau, L. Pulina, E. de la Torre, E. Juarez, P. Muños, S. Salvador, A. Morvan, F. Palumbo, and M. Masin. *Models of Computation.* Tech. rep. D3.5. CERBERO H2020-ICT-2016, June 2018. URL: https://www.cerbero-h2020.eu/wp-content/uploads/2018/12/D3.5.pdf (cit. on p. 19).

[T6] D. Madronal, L. L. Raquel, K. Desnos, F. Arrestier, A. Morvan, M. Pelcat, and E. Juarez. *Dataflow applications real-time monitoring PoC: PREESM-SPIDER-PAPIFY/PAPIFY-Viewer.* Tech. rep. CERBERO H2020-ICT-2016, 2018. URL: https://youtu.be/9QbqtEjKI2U.

[T7] M. Pelcat, K. Desnos, L. Maggiani, Y. Liu, J. Heulot, J.-F. Nezan, and S. S. Bhattacharyya. *Models of Architecture.* Research Report PREESM/2015-12TR01, 2015. INSA Rennes, Dec. 2015.

[T8] P.-Y. Raumer, N. Sourbier, M. Dardaillon, M. Pelcat, and K. Desnos. *Reinforcement Learning Library based on Tangled Program Graphs: Development of New Learning Environments and Library Features.* Tech. rep. INSA Rennes, Aug. 2020. URL: http://kdesnos.fr/wp-content/uploads/publis/20200831_tech_report.pdf.

Bibliography

[1] Brown T. et al. “Language Models are Few-Shot Learners”. In: (2020). arXiv: 2005.14165 [cs.CL] (cit. on p. 61).

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (cit. on pp. 2, 78).

[3] O. Assare and R. K. Gupta. “Performance Analysis of Timing-Speculative Processors”. In: *IEEE Transactions on Computers* 71.2 (2022), pp. 407–420 (cit. on p. 46).

[4] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. “StarPU: a unified platform for task scheduling on heterogeneous multicore architectures”. In: *Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings 15*. Springer. 2009, pp. 863–874 (cit. on pp. 76, 78).

[5] M. Barr. “Real men program in C”. In: *Embedded systems design* 22.7 (2009), p. 3 (cit. on p. 6).

[6] A. Bastidas Fuertes, M. Pérez, and J. Meza Hormaza. “Transpilers: A Systematic Mapping Review of Their Usage in Research and Industry”. In: *Applied Sciences* 13.6 (2023), p. 3667 (cit. on p. 11).

[7] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. “The Arcade Learning Environment: An Evaluation Platform for General Agents”. In: *CoRR* (2012). eprint: 1207.4708. URL: http://arxiv.org/abs/1207.4708 (cit. on pp. 65, 72).

[8] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet. “Overview of the MPEG reconfigurable video coding framework”. In: *Journal of Signal Processing Systems* 63 (2011), pp. 251–263 (cit. on p. 78).

[9] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. “APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations”. In: *Design Automation for Embedded Systems* 2 (1997), pp. 33–60 (cit. on pp. 33, 34).

[10] E. Blossom. “GNU radio: tools for exploring the radio frequency spectrum”. In: *Linux journal* 2004.122 (2004), p. 4 (cit. on p. 2).

[11] J. Bonnot. “Error Analysis for Approximate Computing Systems”. PhD thesis. Rennes, INSA, 2019 (cit. on pp. 45, 54).

[12] P. C. Broekema, R. V. Van Nieuwpoort, and H. E. Bal. “Exascale high performance computing in the square kilometer array”. In: *Proceedings of the 2012 workshop on High-Performance Computing for Astronomy Date*. 2012, pp. 9–16 (cit. on p. 76).

[13] A. Canziani, A. Paszke, and E. Culurciello. “An analysis of deep neural network models for practical applications”. In: *arXiv preprint* (2016). URL: https://arxiv.org/pdf/1605.07678.pdf (cit. on p. 61).

[14] J. Castrillon, M. Lieber, S. Klüppelholz, M. Völp, N. Asmussen, U. Assmann, F. Baader, C. Baier, G. Fettweis, J. Fröhlich, et al. “A hardware/software stack for heterogeneous systems”. In: *IEEE Transactions on Multi-Scale Computing Systems* 4.3 (2017), pp. 243–259 (cit. on p. 4).

[15] M. Chui, M. Collins, and M. Pattel. *The Internet of Things:Catching up to anaccelerating opportunity.* Tech. rep. McKinsey & Compagny, Nov. 2021 (cit. on p. 6).

[16] P. Ciambra, M. Dardaillon, M. Pelcat, and H. Yviquel. “Co-optimizing Dataflow Graphs and Actors with MLIR”. In: *2022 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE. 2022, pp. 1–6 (cit. on p. 78).

[17] CISCO. *Cisco Global Cloud Index: Forecast and Methodology, 2016 – 2021.* White Paper. 2018. URL: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.pdf (cit. on pp. 55, 61).

[18] N. Cressie. “The origins of kriging”. In: *Mathematical geology* 22 (1990), pp. 239–252 (cit. on p. 53).

[19] N. Das, M. Shanbhogue, S.-T. Chen, F. Hohman, S. Li, L. Chen, M. E. Kounavis, and D. H. Chau. “Shield: Fast, practical defense and vaccination for deep learning using jpeg compression”. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2018, pp. 196–204 (cit. on p. 57).

[20] H. Deroui. “Etude et implantation d’algorithmes pour le placement et l’ordonnancement d’applications Dataflow”. Thèse de doctorat dirigée par Nezan, Jean-François Signal, Image, Vision Rennes, INSA 2019. PhD thesis. 2019. URL: http://www.theses.fr/2019ISAR0022 (cit. on pp. 23, 25).

[21] B. D. de Dinechin. “A Qualitative Approach to Many-core Architecture”. In: *Multi-Processor System-on-Chip 1: Architectures* (2021), pp. 27–52 (cit. on p. 6).

[22] S. Dodge and L. Karam. “Understanding how image quality affects deep neural networks”. In: *2016 eighth international conference on quality of multimedia experience (QoMEX)*. IEEE. 2016, pp. 1–6 (cit. on p. 57).

[23] M. Duranton et al. *Key recommendations of the HiPEAC Vision 2021*. Tech. rep. European Network on High-performance Embedded Architecture and Compilation (HiPEAC), 2021. URL: https://www.hipeac.net/media/public/vision/article/HiPEAC_Vision_2021_00.pdf (cit. on p. 12).

[24] W. Ecker, W. Müller, and R. Dömer. “Hardware-dependent software”. In: *Hardware-dependent Software*. Springer, 2009, pp. 1–13 (cit. on p. 20).

[25] N. Edu. *How Many Decimals of Pi Do We Really Need?* online. Oct. 2022. URL: https://www.jpl.nasa.gov/edu/news/2016/3/16/how-many-decimals-of-pi-do-we-really-need/ (cit. on p. 44).

[26] P. Fradet, A. Girault, and P. Poplavko. “SPDF: A schedulable parametric dataflow MoC”. In: *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2012, pp. 769–774 (cit. on p. 11).

[27] N. Gac, J.-F. Nezan, A. Ferrari, C. Ferrari, M. Quinson, and C. Dumez-Viou. *Rapid prototyping of a supercomputer dedicated to radio astronomy*. 2023 (cit. on p. 76).

[28] M. Geilen, T. Basten, and S. Stuijk. “Minimising buffer requirements of synchronous dataflow graphs with model checking”. In: *Proceedings of the 42nd annual Design Automation Conference*. 2005, pp. 819–824 (cit. on p. 11).

[29] A. H. Ghamarian, S. Stuijk, T. Basten, M. Geilen, and B. D. Theelen. “Latency minimization for synchronous data flow graphs”. In: *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*. IEEE. 2007, pp. 189–196 (cit. on p. 11).

[30] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer. “A survey of quantization methods for efficient neural network inference”. In: *Low-Power Computer Vision*. Chapman and Hall/CRC, 2022, pp. 291–326 (cit. on p. 80).

[31] S. Ghosh-Dastidar and H. Adeli. “Spiking neural networks”. In: *International journal of neural systems* 19.04 (2009), pp. 295–308 (cit. on p. 81).

[32] Open Hardware Group. *CORE-V Family of Open-Source RISC-V Cores*. online. Sept. 2022. URL: https://github.com/openhwgroup/core-v-cores (cit. on p. 5).

[33] GrowthEnabler. *Market Pulse Report, Internet of Things*. Tech. rep. 2017. URL: https://growthenabler.com/flipbook/pdf/IOT%20Report.pdf (cit. on p. 54).

[34] Y. Han, G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang. “Dynamic neural networks: A survey”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.11 (2021), pp. 7436–7456 (cit. on p. 78).

[35] J. Hascoët, B. D. de Dinechin, P. G. de Massas, and M. Q. Ho. “Asynchronous one-sided communications and synchronizations for a clustered manycore processor”. In: *Proceedings of the 15th IEEE/ACM Symposium on Embedded Systems for Real-Time Multimedia*. 2017, pp. 51–60 (cit. on p. 10).

[36] N.-M. Ho and W.-F. Wong. “Exploiting half precision arithmetic in Nvidia GPUs”. In: *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2017, pp. 1–7 (cit. on p. 46).

[37] A. Honorat, M. Dardaillon, H. Miomandre, and J.-F. Nezan. “Automated Buffer Sizing of Dataflow Applications in a High-Level Synthesis Workflow”. In: *ACM Transactions on Reconfigurable Technology and Systems* 17.1 (2024), pp. 1–26 (cit. on p. 77).

[38] Q. Hou, K. Zhou, and B. Guo. “BSGP: bulk-synchronous GPU programming”. In: *ACM Transactions on Graphics (TOG)* 27.3 (2008), pp. 1–12 (cit. on p. 76).

[39] Intel. *Xeon Platinum 9282 Processos*. [online]. 2019. URL: https://ark.intel.com/content/www/us/en/ark/products/194146/intel-xeon-platinum-9282-processor-77m-cache-2-60-ghz.html (cit. on p. 6).

[40] H. Jiang, F. J. H. Santiago, H. Mo, L. Liu, and J. Han. “Approximate arithmetic circuits: A survey, characterization, and recent applications”. In: *Proceedings of the IEEE* 108.12 (2020), pp. 2108–2135 (cit. on p. 46).

[41] X. Jiao, V. Camus, M. Cacciotti, Y. Jiang, C. Enz, and R. K. Gupta. “Combining structural and timing errors in overclocked inexact speculative adders”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. Ieee. 2017, pp. 482–487 (cit. on pp. 46, 49).

[42] N. Jouppi, C. Young, N. Patil, and D. Patterson. “Motivation for and Evaluation of the First Tensor Processing Unit”. In: *IEEE Micro* 38.3 (2018), pp. 10–19 (cit. on p. 6).

[43] G. Kahn. “The semantics of a simple language for parallel programming”. In: *Information processing* 74 (1974), pp. 471–475 (cit. on p. 20).

[44] S. Kanur, J. Lilius, and J. Ersfolk. “Detecting data-parallel synchronous dataflow graphs”. In: *2017 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE. 2017, pp. 1–6 (cit. on p. 77).

[45] J. Keinert and E. F. Deprettere. “Multidimensional dataflow graphs”. In: *Handbook of Signal Processing Systems*. Springer, 2013, pp. 1145–1175 (cit. on p. 79).

[46] S. Kelly. “Scaling genetic programming to challenging reinforcement tasks through emergent modularity”. PhD thesis. Halifax, Nova Scotia, Canada: Dalhousie University, 2018. URL: https://dalspace.library.dal.ca/bitstream/handle/10222/73979/Kelly-Stephen-PhD-CSCI-June-2018.pdf (cit. on pp. 63–66, 72).

[47] S. Kelly and M. I. Heywood. “Emergent tangled graph representations for Atari game playing agents”. In: *Genetic Programming: 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings 20*. Springer. 2017, pp. 64–79 (cit. on pp. 46, 62, 63, 65, 66, 69).

[48] S. Kelly, J. Newsted, W. Banzhaf, and C. Gondro. “A Modular Memory Framework for Time Series Prediction”. In: *Genetic and Evolutionary Computation Conference*. GECCO ’20. Cancún, Mexico: Association for Computing Machinery, 2020, pp. 949–957. URL: https://doi.org/10.1145/3377930.3390216 (cit. on pp. 63, 66).

[49] F. Kermarrec, S. Bourdeauducq, H. Badier, and J.-C. Le Lann. “LiteX: an open-source SoC builder and library based on Migen Python DSL”. In: *OSDA 2019, colocated with DATE 2019 Design Automation and Test in Europe*. 2019 (cit. on p. 5).

[50] P. M. Knijnenburg, T. Kisuki, and M. F. O’Boyle. “Iterative compilation”. In: *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation—SAMOS* (2002), pp. 171–187 (cit. on p. 79).

[51] H.-J. Ko and J. J. Tsai. “Robust and computationally efficient digital IIR filter synthesis and stability analysis under finite precision implementations”. In: *IEEE Transactions on Signal Processing* 68 (2020), pp. 1807–1822 (cit. on p. 47).

[52] A. Krizhevsky, G. Hinton, et al. “Learning multiple layers of features from tiny images”. In: (2009) (cit. on p. 44).

[53] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., 2012, pp. 1097–1105 (cit. on pp. 6, 7, 44, 46, 57, 61, 66).

[54] A. Kuusela and C. Smullen. “Video Coding Unit (VCU): Hot Chips 2021”. In: *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE. 2021, pp. 1–30 (cit. on p. 6).

[55] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. “MLIR: Scaling compiler infrastructure for domain specific computation”. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2021, pp. 2–14 (cit. on p. 78).

[56] R. Lazcano, D. Madroñal, E. Juarez, and P. Clauss. "Runtime multi-versioning and specialization inside a memoized speculative loop optimizer". In: *Proceedings of the 29th International Conference on Compiler Construction*. 2020, pp. 96–107 (cit. on p. 16).

[57] E. A. Lee and S. Ha. "Scheduling strategies for multiprocessor real-time DSP". In: *1989 IEEE Global Telecommunications Conference and Exhibition'Communications Technology for the 1990s and Beyond'*. IEEE. 1989, pp. 1279–1283 (cit. on p. 10).

[58] E. A. Lee and I. John. "Overview of the ptolemy project". In: (1999) (cit. on p. 78).

[59] E. A. Lee and D. G. Messerschmitt. "Synchronous data flow". In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245 (cit. on pp. 8, 20–22, 25, 29).

[60] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang. "Pruning and quantization for deep neural network acceleration: A survey". In: *Neurocomputing* 461 (2021), pp. 370–403 (cit. on p. 46).

[61] C. Liu, J. Han, and F. Lombardi. "An analytical framework for evaluating the error characteristics of approximate adders". In: *IEEE Transactions on Computers* 64.5 (2014), pp. 1268–1281 (cit. on p. 49).

[62] T. Liu and S.-L. Lu. "Performance improvement with circuit-level speculation". In: *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. 2000, pp. 348–355 (cit. on p. 49).

[63] C. Lomont. "Fast inverse square root". In: *Tech-315 nical Report* 32 (2003) (cit. on pp. 44, 46).

[64] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas. "Bio-inspired imprecise computational blocks for efficient VLSI implementation of soft-computing applications". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 57.4 (2009), pp. 850–862 (cit. on p. 49).

[65] Q. Milot, M. Dardaillon, J. Bonnot, and D. Ménard. "Wordlength Optimization for Custom Floating-Point Systems". In: *Workshop on Design and Architectures for Signal and Image Processing (DASIP)*. LNCS. Munich, Germany: Springer, 2024 (cit. on p. 46).

[66] H. Miomandre, J.-F. Nezan, D. Menard, A. Campbell, A. Griffin, S. Hall, and A. Ensor. "Approximate buffers for reducing memory requirements: Case study on SKA". In: *2020 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE. 2020, pp. 1–6 (cit. on pp. 9, 48).

[67] G. E. Moore et al. *Cramming more components onto integrated circuits*. 1965 (cit. on p. 4).

[68] K. Muvva, J. M. Bradley, M. Wolf, and T. Johnson. "Assuring learning-enabled components in small unmanned aircraft systems". In: *AIAA Scitech 2021 Forum*. 2021, p. 0994 (cit. on p. 44).

[69] S. Neuendorffer and E. Lee. “Hierarchical reconfiguration of dataflow models”. In: *International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE’04.* IEEE and ACM. 2004, pp. 179–188 (cit. on p. 22).

[70] E. Nogues, D. Menard, and M. Pelcat. “Algorithmic-level approximate computing applied to energy efficient hevc decoding”. In: *IEEE Transactions on Emerging Topics in Computing* 7.1 (2016), pp. 5–17 (cit. on p. 46).

[71] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks. “Vapor SIMD: Auto-vectorize once, run everywhere”. In: *International Symposium on Code Generation and Optimization (CGO 2011).* IEEE. 2011, pp. 151–160 (cit. on p. 79).

[72] K. Ota, M. S. Dao, V. Mezaris, and F. G. D. Natale. “Deep learning for mobile multimedia: A survey”. In: *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 13.3s (2017), pp. 1–22. URL: `https://dl.acm.org/doi/abs/10.1145/3092831` (cit. on p. 61).

[73] F. Palumbo, N. Carta, D. Pani, P. Meloni, and L. Raffo. “The multi-dataflow composer tool: generation of on-the-fly reconfigurable platforms”. In: *Journal of real-time image processing* 9 (2014), pp. 233–249 (cit. on p. 16).

[74] K. Parashar, R. Rocher, D. Menard, and O. Sentieys. “A hierarchical methodology for word-length optimization of signal processing systems”. In: *2010 23rd International Conference on VLSI Design.* IEEE. 2010, pp. 318–323 (cit. on p. 53).

[75] M. Pelcat. *GHG emissions of semiconductor manufacturing in 2021.* Tech. rep. Univ Rennes, INSA Rennes, CNRS, IETR – UMR 6164, F-35000 Rennes, June 2023. URL: `https://hal.science/hal-04112708` (cit. on p. 8).

[76] J. Piat, S. S. Bhattacharyya, and M. Raulet. “Interface-based hierarchy for synchronous data-flow graphs”. In: *2009 IEEE Workshop on Signal Processing Systems.* 2009, pp. 145–150 (cit. on p. 22).

[77] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee. “A hierarchical multiprocessor scheduling system for DSP applications”. In: *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers.* Vol. 1. IEEE. 1995, pp. 122–126 (cit. on p. 33).

[78] M. Poorhosseini, W. Nebel, and K. Grüttner. “A Compiler Comparison in the RISC-V Ecosystem”. In: *2020 International Conference on Omni-layer Intelligent Systems (COINS).* 2020, pp. 1–6 (cit. on p. 5).

[79] H. A. M. Puat and N. A. Abd Rahman. “IoMT: a review of pacemaker vulnerabilities and security strategy”. In: *Journal of Physics: Conference Series.* Vol. 1712. 1. IOP Publishing. 2020, p. 012009 (cit. on p. 7).

[80] A. Rahimi, A. Ghofrani, K.-T. Cheng, L. Benini, and R. K. Gupta. “Approximate associative memristive memory for energy-efficient GPUs”. In: *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE. 2015, pp. 1497–1502 (cit. on p. 46).

[81] V. Rajagopalan, V. Boppana, S. Dutta, B. Taylor, and R. Wittig. "Xilinx Zynq-7000 EPP: An extensible processing platform family". In: *2011 IEEE Hot Chips 23 Symposium (HCS)*. 2011, pp. 1–24 (cit. on p. 5).

[82] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener. "Programming with relaxed synchronization". In: *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*. 2012, pp. 41–50 (cit. on p. 46).

[83] M. Rocklin et al. "Dask: Parallel computation with blocked algorithms and task scheduling". In: *Proceedings of the 14th python in science conference*. Vol. 130. SciPy Austin, TX. 2015, p. 136 (cit. on p. 78).

[84] A. Roelke and M. R. Stan. "Risc5: Implementing the RISC-V ISA in gem5". In: *First Workshop on Computer Architecture Research with RISC-V (CARRV)*. Vol. 7. 17. 2017 (cit. on p. 5).

[85] G. Roumage, S. Azaiez, and S. Louise. "A survey of main dataflow MoCCs for CPS design and verification". In: *2022 IEEE 15th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE. 2022, pp. 1–9 (cit. on pp. 11, 19, 26).

[86] F. S. Saadatmand, T. Stefanov, I. G. Alonso, A. D. Pimentel, B. Akesson, M. Herget, and M. Bor. "Automated Derivation of Application Workload Models for Design Space Exploration of Industrial Distributed Cyber-Physical Systems". In: *IEEE International Conference on Industrial Cyber-Physical Systems (ICPS)*. 2024 (cit. on p. 32).

[87] P. K. Sahu, S. R. Pal, and A. K. Das. *Estimation and inferential statistics*. Springer, 2015 (cit. on p. 51).

[88] J. E. Savage. *Models of computation*. Vol. 136. Addison-Wesley Reading, MA, 1998 (cit. on pp. 3, 19).

[89] T. Schwarzer, J. Falk, S. Müller, M. Letras, C. Heidorn, S. Wildermann, and J. Teich. "Compilation of dataflow applications for multi-cores using adaptive multi-objective optimization". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 24.3 (2019), pp. 1–23 (cit. on p. 39).

[90] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou. "Memory devices and applications for in-memory computing". In: *Nature nanotechnology* 15.7 (2020), pp. 529–544 (cit. on pp. 5, 10).

[91] O. Sentieys and D. Menard. "Customizing Number Representation and Precision". In: *Approximate Computing Techniques: From Component-to Application-Level*. Springer, 2022, pp. 11–41 (cit. on p. 46).

[92] J. Sérot. "Hocl: High level specification of dataflow graphs". In: *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages*. 2020, pp. 11–22 (cit. on p. 78).

[93] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra. "A review of near-memory computing architectures: Opportunities and challenges". In: *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE. 2018, pp. 608–617 (cit. on p. 5).

[94] R. J. Smith, R. Amaral, and M. I. Heywood. "Evolving simple solutions to the CIFAR-10 benchmark using tangled program graphs". In: *2021 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2021, pp. 2061–2068 (cit. on p. 66).

[95] IEEE Spectrum. *Top Programming Languages 2022*. [online]. Aug. 2022. URL: https://spectrum.ieee.org/top-programming-languages-2022 (cit. on p. 6).

[96] B. Stabernack and F. Steinert. "Architecture of a Low Latency H.264/AVC Video Codec for robust ML based Image Classification". In: *Workshop on Design and Architectures for Signal and Image Processing (14th Edition)*. DASIP '21. Budapest, Hungary: Association for Computing Machinery, 2021, pp. 1–9. URL: https://doi.org/10.1145/3441110.3441149 (cit. on pp. 56, 59).

[97] S. Stuijk, M. Geilen, and T. Basten. "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs". In: *Proceedings of the 43rd annual design automation conference*. 2006, pp. 899–904 (cit. on pp. 11, 12).

[98] H. Svenshon. "Heron of Alexandria and the dome of Hagia Sophia in Istanbul". In: *Proceedings of the Third International Congress on Construction History*. Vol. 3. 2009, pp. 1387–94 (cit. on p. 44).

[99] M. Tan and Q. Le. "Efficientnet: Rethinking model scaling for convolutional neural networks". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 6105–6114 (cit. on p. 62).

[100] B. Tippetts, D. J. Lee, K. Lillywhite, and J. Archibald. "Review of stereo vision algorithms and their suitability for resource-limited systems". In: *Journal of Real-Time Image Processing* 11 (2016), pp. 5–25 (cit. on p. 9).

[101] E. Vasilakis. "An instruction level energy characterization of arm processors". In: *Foundation of Research and Technology Hellas, Inst. of Computer Science, Tech. Rep. FORTH-ICS/TR-450* (2015) (cit. on p. 45).

[102] G. Venkatesh, E. Nurvitadhi, and D. Marr. "Accelerating deep convolutional networks using low-precision and sparsity". In: *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2017, pp. 2861–2865. URL: https://ieeexplore.ieee.org/abstract/document/7952679 (cit. on p. 61).

[103] A. Waterman, Y. Lee, D. Patterson, K. Asanovic, V. I. U. level Isa, A. Waterman, Y. Lee, and D. Patterson. "The RISC-V instruction set manual". In: *Volume I: User-Level ISA', version* 2 (2014) (cit. on p. 4).

[104] M. Wipliez and M. Raulet. “Classification and transformation of dynamic dataflow programs”. In: *2010 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE. 2010, pp. 303–310 (cit. on p. 78).

[105] N. Wirth. “A plea for lean software”. In: *Computer* 28.2 (1995), pp. 64–68 (cit. on p. 7).

[106] C. Wolf. *Picorv32-a size-optimized RISC-V cpu (2019)*. 2019. URL: `https://github.com/YosysHQ/picorv32` (cit. on p. 5).

[107] H.-S. P. Wong, R. Willard, and I. K. Bell. “IC Technology – What Will the Next Node Offer Us?” In: *2019 IEEE Hot Chips 31 Symposium (HCS)*. 2019, pp. 1–52 (cit. on p. 4).

[108] C. Wu, R. Tobar, K. Vinsen, A. Wicenec, D. Pallot, B.-q. Lao, R. Wang, T. An, M. Boulton, I. Cooper, et al. “DALiuGE: A graph execution framework for harnessing the astronomical data deluge”. In: *Astronomy and computing* 20 (2017), pp. 1–15 (cit. on p. 2).

[109] W. A. Wulf and S. A. McKee. “Hitting the memory wall: implications of the obvious”. In: *ACM SIGARCH* 23.1 (1995), pp. 20–24 (cit. on p. 5).

[110] S.-J. Yen and Y.-S. Lee. “Under-sampling approaches for improving prediction of the minority class in an imbalanced dataset”. In: *Intelligent Control and Automation: International Conference on Intelligent Computing, ICIC 2006 Kunming, China, August 16–19, 2006*. Springer. 2006, pp. 731–740 (cit. on p. 74).

[111] C. Yu, S. Royuela, and E. Quiñones. “Taskgraph: A Low Contention OpenMP Tasking Framework”. In: *IEEE Transactions on Parallel and Distributed Systems* 34.8 (2023), pp. 2325–2336 (cit. on p. 76).

[112] S. Zheng, Y. Song, T. Leung, and I. Goodfellow. “Improving the robustness of deep neural networks via stability training”. In: *Proceedings of the ieee conference on computer vision and pattern recognition*. 2016, pp. 4480–4488 (cit. on p. 57).

[113] W. Banzhaf, P. Machado, and M. Zhang, eds. *Handbook of Evolutionary Machine Learning*. Springer, 2023 (cit. on p. 63).

Titre : Automatisation de la conception et optimisations multi-objective pour logiciel embarqués à haute performance

Mot clés : Conception automatisée, Systèmes embarqués, Flux de données, Calculs approximés, IA frugale

Résumé : Ce manuscrit d'habilitation présente dix ans de recherche sur la conception des systèmes embarqués de haute performance selon trois axes de recherche.

Le premier axe porte sur l'utilisation de modèles de flux de données de calcul pour programmer des puces hétérogènes multicœurs. Des extensions de la sémantique du flux de données ont été proposées, ainsi que des optimisations basées modèles pour automatiser le déploiement d'applications sur des systèmes de calcul à haute performance.

Le deuxième axe concerne l'utilisation d'approximations pour réduire la complexité de calcul des systèmes complexes. Des méthodes de caractérisation rapide des erreurs causées par les approximations ont été développées, ainsi que des techniques pour exploiter la résilience des algorithmes de vision artificielle aux artefacts de la compression d'images avec perte.

Le troisième axe concerne le développement d'agents artificiels frugaux basés sur les graphes de programmes entremêlés (TPGs). Des contributions ont été proposées pour accélérer l'apprentissage des TPGs grâce à la parallélisation, et pour accélérer l'inférence pour les dispositifs à faible puissance grâce à la génération d'un code d'inférence dédié.

Le dernier chapitre du manuscrit présente les perspectives de recherche ouvertes, notamment l'exploration multi-objectifs et multi-contraintes de l'espace de conception.

Title: Design Automation and Multi-Objective Optimizations for High-Performance Embedded Software

Keywords: Design Automation, Embedded Systems, Dataflow, Approximate Computing, Frugal AI

Abstract: This habilitation manuscript presents a decade of research aimed at taming the design complexity of high-performance embedded systems, with contributions along three main research axes.

The first research axis explored is the use of dataflow models of computation to program heterogeneous multicore chips. Contributions include extensions to dataflow semantics and model-based optimizations to automate the deployment of large applications on high-performance computing systems.

The second research axis explores the use of approximations as a tool to reduce the computational complexity of complex systems. Proposed contributions include generic methods for rapidly characterizing hardware and software errors caused by approximations, and techniques for exploiting the resilience of computer vision algorithms to artifacts introduced by lossy image compression.

The third research axis explored concerns the development of frugal artificial agents based on Tangled Program Graphs (TPGs). Contributions in this area aim at accelerating the training of TPGs through parallelization, and at accelerating inference for low-power devices through the generation of dedicated inference code.

Research perspectives that follow the ongoing shift towards multi-objective and multi-constraint design space exploration are presented in the final chapter of the manuscript.