TESI DI DOTTORATO

UNIVERSITÀ DEGLI STUDI DI NAPOLI "FEDERICO II"

DIPARTIMENTO DI INGEGNERIA ELETTRONICA
E DELLE TELECOMUNICAZIONI

DOTTORATO DI RICERCA IN
INGEGNERIA ELETTRONICA E DELLE
TELECOMUNICAZIONI

# DESIGNING EFFICIENT COMPUTING SYSTEMS: THE APPROXIMATE-COMPUTING BREAKTHROUGH

**SALVATORE BARONE**

Il Coordinatore del Corso di Dottorato
Ch.mo Prof. Daniele RICCIO

Il Tutore
Ch.mo Prof. Antonino MAZZEO

A. A. 2020–2021

# Abstract

Approximate Computing (AxC) paradigm aims at designing computing systems that can satisfy the rising performance demands and improve the energy efficiency. AxC exploits the gap between the level of accuracy required by a given application, and the actual precision provided by the computing system, for achieving diverse optimizations. Various AxC techniques have been proposed so far in the literature at different abstraction levels from hardware to software. These techniques have been successfully utilized and combined to realize approximate implementations of applications in various domains (e.g., data analytic, scientific computing, multimedia and signal processing, and machine learning). Unfortunately, state-of-the-art approximation methodologies focus on a single abstraction level, such as combining elementary components, e.g., arithmetic operations, and usually optimize hardware-requirements under error constraints, resulting in suboptimal solutions. This hinders the possibility for designers to explore different approximation opportunities, optimized for different applications and implementation targets. Therefore, we propose a methodology for the design of approximate applications which is based on multi-objective optimization and does not depend on either applications or techniques. We discuss each phase and steps the methodology breaks into, while devoting the needed relevance to their automation. In order to validate and evaluate our method, we resort to a vast plethora of applications, including generic combinational logic, arithmetic circuits, image-processing and machine-learning applications. For each of them, we report several case studies and experiments that empirically prove the validity and effectiveness of the methodology, which allow achieving significant savings both in terms of area and power required by hardware accelerators, at the cost of very low introduced error.

# Preface

Some of the research and results described in this Ph.D. thesis has undergone peer-review and has been published in, or at the date of this printing is being considered for publication in, academic journals and books. In the following, I list the papers developed during my research work as Ph.D. student.

1. M. Barbareschi, S. Barone, N. Mazzocca. Advancing synthesis of decision tree-based multiple classifier systems: an approximate computing case study. Knowledge and Information Systems 63 (6), 1577-1596, 2021.
2. S. Barone, M. Traiola, M. Barbareschi, A. Bosio. Multi-Objective Application-driven Approximate Design Method. IEEE Access, 2021.
3. M. Barbareschi, S. Barone, A. Bosio, M. Traiola, J. Han. A Genetic-Algorithm-Based Approach to the Design of DCT Hardware Accelerators. ACM Journal on Emerging Technologies in Computing Systems.
4. M. Barbareschi, S. Barone, N. Mazzocca, and A. Moriconi. A Catalog-based AIG-Rewriting Approach to the Design of Approximate Components. IEEE Transaction on Emerging Topics in Computing. (Currently under review, round two)
5. M. Barbareschi, S. Barone, N. Mazzocca, and A. Moriconi. Design Space Exploration Tools (Book Chapter). (Currently under review)

# Contents

# List of Figures

# List of Tables

# Acronyms

**AIG** And-Inverter Graph. 5, 41–47, 49–54, 56, 161

**ALU** Arithmetic Logic Unit. 100

**AMA** Approximate Mirror Adder. 75, 80

**AMOSA** Archived Multi-Objective Simulated Annealing. 30, 34, 35, 39, 42, 50, 52

**ANN** Artificial Neural Network. 90, 93, 99, 100, 103

**ASIC** Application-Specific Integrated Circuit. xvii, 81–85, 100–102, 162

**AST** Abstract Syntax Tree. xiv, 28, 62, 63, 67, 72, 75, 107

**AWCE** Absolute Worst-Case Error. xiii, xvii, 51, 54, 55, 57, 58

**AXA** Approximate XOR-based Adder. 75, 80

**AxC** Approximate Computing. 2–5, 7–9, 13, 25–27, 38, 59–61, 66, 89, 90, 103, 158, 161, 162

**B&B** Branch & Bound. 152, 160

**BDD** Binary Decision Diagram. 16

**BMF** Boolean Matrix Factorization. 22

**CL** Convolutional Layer. 91, 92, 99, 102, 108, 109, 111, 112, 115, 116, 120, 128–131

**CMOS** Complementary Metal-Oxide Semiconductor. 86

**CNN** Convolutional Neural Network. 91–94, 99, 105, 108, 111, 120, 124, 126, 162

**CPG** Cartesian Genetic Programming. 22, 23, 105

**DBX** Decision-BoX. 147

**DCT** Discrete Cosine Transform. xiv, xvii, 5, 13, 15, 59, 71–80, 82, 83, 86, 87, 162

**DFG** Data-Flow Graph. 24

**DNN** Deep Neural Network. xiv, 6, 27, 38, 89, 91, 92, 94, 98, 99, 102, 103, 106, 107, 110, 151, 162

**DSE** Design-Space Exploration. xiii, xvii, xviii, 4, 5, 7–9, 24–27, 29, 30, 38, 39, 41, 42, 50–52, 55, 57, 58, 61, 68, 69, 72, 77, 78, 80, 81, 83, 87, 90, 106, 107, 110, 111, 113–118, 120, 127, 130, 131, 152–154, 161, 162

**DSP** Digital Signal Processing. 14, 15, 82, 114

**DSSIM** Structural DISSIMilarity. 77, 84–86

**DT** Decision Tree. xv, 132–134, 138, 140–142, 146, 147, 149, 154, 155, 158, 159

**DT MCS** Decision Tree based Mutiple Classifier System. xvi, 6, 27, 38, 89, 151–154, 159, 162, 163

**E-𝕀DE𝔸** Evolutionary-IIDEAA Is a Design Exploration tool for Approximate Algorithm. xiii, 59–62, 68, 69, 75, 106, 152, 154

**EA** Evolutionary Algorithm. 30, 33

**ES** Exact Synthesis. 42–47, 49–51

**EU** Energy Unit. 115

**ExDC** External Don't Care. 20, 21

**FAC** Full-Adder Cell. 72, 75–77, 79, 80, 82, 83

**FCL** Fully-Connected Layer. 99, 102, 108, 109, 111, 112, 116, 120, 128, 129

**FFT** Fast Fourier Transform. 15

**FinFET** Fin Field-Effect Transistor. xiv, 81–83, 125

**FIR** Finite Impulse Response. 15

**FLAP** FLexible Arithmetic Precision. 62

**FPGA** Field Programmable Gate Array. xiv, xv, 46, 52, 55, 57, 81–85, 100–102, 107, 109, 113–117, 120–122, 141, 142, 145, 152, 155, 159, 162

**FPU** Floating-Point Unit. 12

**FRAIG** Functionally-Reduced And-Inverter Graph. 51

**GA** Genetic Algorithm. 30–32, 69, 70, 113, 153, 162

**GMDH** Group Method of Data Handling. 90

**GP-GPU** General Purpose - Graphic Processing Unit. 12, 92, 93, 100, 141

**HDL** Hardware Description Language. 50

**HLS** High Level Synthesis. 61

**IAC** Inexact-Adder Cell. 72, 75–77, 79, 80, 82, 83, 86

**IJCNN** International Joint Conference on Neural Networks. 93

**ILP** Integer Linear Programming. 16

**InXA** IneXact Adder. 75, 80

**LLVM** Low Level Virtual Machine. 62, 66

**LUT** Look-Up Table. xiii–xv, 46, 47, 49–51, 53, 54, 82–85, 106, 114, 115, 117, 118, 120, 121, 155, 158–160

**MNIST** Modified National Institute of Standards and Technology. 93, 111, 124

**MOEA** Multi-Objective Evolutionary Algorithm. xiv, 77, 81, 86, 87

**MOP** Multi-objective Optimization Problem. 4, 5, 9, 22, 25, 27, 29–31, 35, 38, 39, 41, 42, 50, 55–58, 65, 66, 72, 76, 90, 106, 111, 115, 117, 124, 130, 131, 152, 153, 160–162

**MOSFET** Metal-Oxide-Semiconductor Field-Effect Transistor. 1

**MP-CNN** Max-Pooling Convolutional Neural-Network. 92, 93

**MPL** Max-Pooling Layer. 92

**MSSIM** Mean SSIM. 76, 77

**NAB** Number of Approximate Bit. 28, 64, 66, 75–79, 86, 110, 111, 120

**NoC** Network on Chip. 102

**NSGA-II** Non-dominated Sorting Genetic Algorithm-II. xiii, 23, 32, 33, 39, 66, 71, 72, 80, 107, 113, 116, 120, 152, 160, 163

**ODC** Observability Don't Care. 20, 21

**PE** Processing Element. xiv, 100, 101, 109, 110, 122

**PI** Primary Input. 21, 42–44, 46, 51

**PL** Pooling Layer. 91, 99, 108

**PO** Primary Output. 21, 43

**PSNR** Peak Signal-to-Noise Ratio. xiv, 3, 38, 68–70, 86, 87

**QCC** Quality Constraint Circuit. 20

**QEC** Quality Evaluation Circuit. 21

**ReLU** Rectifier Linera Unit. 93, 98, 101

**RISC** Reduced Instruction-Set Computer. 144

**RMSE** Root Mean Squared Error. 104

**RNN** Recurrent Neural Network. 92, 94, 99

**RTL** Register-Transfer Level. 20, 23, 78, 109, 142, 145

**SA** Simulated Annealing. 34

**SAT** Boolean SATisfiability. 16, 42, 44

**SIMD** Single-Instruction-Multiple-Data. 13, 141

**SIMT** Single-Instruction-Multiple-Thread. 13, 141

**SMT** Satisfiability-Modulo Theory. 44, 49, 51

**SNR** Signal-to-Noise Ratio. 15

**SQCC** Sequential Quality Constraint Circuit. 21, 22

**SSIM** Structural SIMilarity. 3, 38, 76, 77

**VPA** Variable Precision Arithmetic. 62

**WMED** Weighted Mean Error Distance. 24

*"When the night has come*
*And the land is dark*
*And the moon is the only light we'll see*
*No, I won't be afraid*
*Oh, I won't be afraid*
*Just as long as you stand, stand by me"*

*To Francesca,*
*who shared every day with me, even the gloomiest one,*
*without ever letting me lack her love, care, and encouragement.*

# Chapter 1

# Introduction

In the XX century, the human being witnessed an unprecedented amount of world-changing inventions: the airplane, antibiotics, the jet-engine, the nuclear-reactors, and so forth. But, unless you live like an eremite, or you are parts of an undiscovered tribe, there is one single invention that really do changed the life of everyone on the Earth.

Although few seen that directly, it is nowadays the most largely manufactured device in history, as, in 2018, an impressive $10^{22}$ units have been manufactured, with $10^{13}$ new ones manufactured every day, on average [7]. Nothing made by humans comes even close to its manufacturing-rate, neither enabled other world-changing inventions and discoveries. We are evidently talking about the Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET). It is undeniable, in fact, that, looking around us, we can spot at least one electronic component, and that transistors, by now, are literally everywhere.

Their main use, at least for the numbers involved, is, however, the realization of digital computing devices. The first CPU to be commercialized, the Intel 4004 dating back to 1971, was built with a manufacturing process that allowed to integrate 2500 transistors on a silicon wafer of 1 $mm^2$. During that time, Gordon Moore, the co-founder of the Intel corporation, stated what later became known as the Moore's law: "the number of transistors in an integrated circuits doubles every two years". How

to blame him, since with the current 7 nm manufacturing process we can fit over 134 million transistors for each $mm^2$ of silicon [1].

Nevertheless, silicon-die shrinking is approaching its physical limit: the diameter of a single atom of silicon is 0.2 nm, meaning a 7 nm transistor requires only 35 silicon atoms to be manufactured. Moreover, modern computing systems are experimenting an unprecedented growth of data to be processed, since, on one hand, these systems are increasingly used to interact with the physical world and, on the other hand, they process the large amount of data samples coming from all the various sensing sources. Indeed, the scientific literature is mainly focusing on pattern recognition, machine-learning and data-classification systems since, from one side, the Big Data domain gives the possibility to access to heterogeneous and huge datasets, allowing for new applications, from the other side, researches investigate about technological innovations and new methodologies to overcome performance issues, in terms of both model accuracy, throughput, and latency. This lead computing systems requiring a tremendous amount of energy, at an increasing rate every year, so that it is estimated energy consumption will exceed the amount of energy produced before 2040 [58].

Therefore, power and energy reduction are critical requirements in the design of computing systems, especially in pervasive embedded and mobile electronic devices, where the battery capacity is a limiting factor. Additionally, computationally intensive tasks, such as machine-learning applications, have found their way into these power-limited devices, increasing the need for efficient electronics. In this perspective, current technologies and design approaches are bound to become quite soon inadequate to offer suitable solutions to these applications requirements; hence, novel design approaches have to be considered.

Approximate Computing (AxC) term has been introduced to define a computing paradigm applied at different abstraction levels, spanning from hardware to software components. Intuitively, instead of performing exact computation, AxC aims to carefully violate non-critical specifications, trading accuracy off for efficiency. For several real-world scenarios, the effectiveness of imprecise computation has been demonstrated in the literature, for both software and hardware components implementing inexact algorithms. Indeed, some applications show an inherent resiliency to errors [47].

The inherent resiliency property tightly depends on the application domain. It

can be observed for algorithms dealing with noisy real-world input data (e.g., image processing, sensor data processing, speech recognition, etc.), or with outputs that have to be interpreted by humans, such as digital signal processing of images or audio; also data analytic, web search and wireless communications exhibit an equivalent property. Other involved applications are those that iteratively process large amounts of information, sample data, stop the convergence procedure early, or apply heuristics.

Diverse research articles in the literature explored the opportunities provided by AxC paradigm. Many of them proposed new methodologies to automatically define the best trade-off configurations between result quality and performance. However, there are still open challenges holding AxC back from wider employment.

Since a naive approximation approach is unlikely to be efficient, the target application have to be taken into account in order to properly select data or portions of the application to approximate. This may require the designer to have deep knowledge of the application, be fully aware of the target domain. Moreover, a vast plethora of approximate techniques, spanning from software to hardware target implementation, have been proposed in the scientific literature, and selecting the most appropriate one, again, may require a deep understanding of the application being approximate [125]. In addition, complex applications may preclude the manual introduction of approximation, e.g., through source-code annotation, and developing suitable automatic tools is not trivial.

Monitoring the output quality varying the introduced approximation degree is mandatory to guarantee quality-constraints are met [47]. Though, selecting an appropriate error metric, or a set of error metric, is not a trivial task, rather it is of major concern. Some metrics, in fact, are much more sensitive than others. For instance, the PSNR metric, which is commonly adopted in image-processing applications, is quite sensitive to noise, since it consider single image-pixels. This may preclude an effective introduction of approximation w.r.t. some other less-sensitive metrics, such as the Structural SIMilarity (SSIM). Moreover, the choice concerning the error-assessment technique must be well-thought-out because some technique, e.g., exhaustive simulations, may be cumbersome when targeting certain applications, requiring a different technique, e.g., formal methods, to be adopted. Yet, the latter may not be applicable due to the complexity of the applications themselves. When approximating digital logic circuits, for example, exhaustive simulations are prohibitive when the number

of inputs is substantial, so it would be preferable to use a different technique, e.g., formal-methods. However, when more complex applications, such as image-processing, or machine-learning applications, are concerned, the choice on the technique to be used is almost forced to simulations, since modeling such applications is quite cumbersome.

Defining optimal trade-offs between error due to approximation and corresponding savings require addressing conflicting design goals. Introducing low error makes the approximate implementation close to the exact implementation, resulting in low resource savings. These last ones can be conspicuous if the error introduced is substantial. As it easy to recognize, leveraging the AxC requires coping with a Multi-objective Optimization Problem (MOP). Anyway, almost all the approaches from the scientific literature either combine multiple design objectives in one weighted single-objective optimization problem, or optimize for one single parameter while keeping the others fixed, so resulting solutions are centered around a few dominant design alternatives and do not cover the whole Pareto-front [56]. Recent works, however, addressed the circuit design problem by employing MOP to search for Pareto-optimal approximate circuit implementations. Unfortunately, such approaches did not focus on generic applications, rather only on basic arithmetic components, such as adders and multipliers [156].

All the mentioned challenges make defining a generic and application-independent methodology quite difficult, hindering the wide-spread adoption of the AxC.

In this work, we foster an application-independent, unified methodology able to automatically explore the impact of different approximation techniques on a given application, while resorting to the AxC design paradigm and MOP-based DSE. We also devote particular relevance to all the phases and steps of the proposed methodology which can be automated. When compared to contributions from the scientific literature, our methodology is neither tailored to a specific application nor to an approximation technique, it does not require the designer to specify which part(s) of the application should be approximated and how, and it only requires the definition of the acceptable output degradation from the user. Moreover, it addresses the circuit design problem as a MOP, which allows optimizing different figure of metrics, e.g., error and hardware-requirements, at the same time, providing the designer with a set of equally good Pareto-optimal solutions, leaving the designer free to choose the one that, according to his experience, best suits the context or the requirements of the application considered.

In order to validate and evaluate the proposed metrology, we selected some significant and relevant applications in the scope of the AxC paradigm, among which we include generic combinatorial logic, image-processing applications, and artificial intelligence applications.

This Thesis work is organized as follows:

- Chapter 2 provides the reader with a brief introduction concerning the AxC design paradigm, including issues and challenges to be addressed in order to exploit the AxC full potential. In particular, it discusses approaches to identify approximable code portion or data within a given application, common techniques adopted to introduce approximation, methods metrics for assessing error due to approximation, and, finally, how to pick the approximate configurations providing the best trade-offs between introduced error and gains, i.e., how to perform DSE. Moreover, it surveys the state-of-the-art concerning approximation methodologies targeting hardware applications, discussing the adopted approximation technique, the error assessment and DSE approach.

- Chapter 3 discusses our AxC and MOP-based design methodology, including the main steps of the method that we propose, and how the method helps in addressing the challenges that the AxC paradigm poses to the designer. It also devotes particular emphasis to the steps of the methodology that can be automated, including the generation of approximate variants for a give application, and the selection of optimal trade-offs between quality of results and hardware-requirements through the use of a MOP-based DSE.

- Chapter 4 applies the methodology discussed in Chapter 3 addressing the design of combinational logic circuits, i.e., those that typically constitute building-blocks for larger, more complex, designs. In particular, it discusses a novel And-Inverter Graph (AIG)-rewriting based technique to automatically generate approximate variants for combinational circuits, and how to select the best trade-offs between error and savings through the use of a MOP-based DSE.

- Chapter 5 reports the application of our methodology to the design of hardware accelerators for image-processing processing. Specifically, a Sobel edge-detector, and several accelerators for the DCT, which is the most demanding step of

the JPEG, viz. one of the most commonly adopted lossy image and video compression algorithm.

- The last Chapter discusses the application of our methodology on two of the most promising classification models in the machine-learning domain, namely DNNs and DT MCSs. These applications are even more challenging, since hardware-accelerators are utterly resource intensive, and reducing the amount of induced error is very critical, because machine-learning systems process a huge amount of data.

# Chapter 2

# The Approximate Computing Design Paradigm and its Application

This chapter provides the reader with a brief introduction concerning the AxC design paradigm, including issues and challenges to be addressed in order to exploit the AxC full potential. In particular, Section 2.2 discusses approaches to identify approximable code portion or data within a given application in Section 2.2.1, common techniques adopted to introduce approximation in Section 2.2.2, methods metrics for assessing error due to approximation in Section 2.2.3, and, finally, how to pick the approximate configurations providing the best trade-offs between introduced error and gains, i.e., how to perform DSE, in Section 3.1.2. Moreover, Section 2.3 surveys the state-of-the-art concerning approximation methodologies targeting hardware applications, discussing the adopted approximation technique, the error assessment and DSE approach.

## 2.1 Overview

The scientific literature demonstrated that inexact computation can be selectively exploited to enhance computing system performance, defining the AxC paradigm [180]. It is based on the intuitive observation that, while performing exact computation, or maintaining peak-level service performance, require a high amount of resources, allowing selective approximation or occasional violation of the specification can provide quite interesting gains in efficiency. For example, for a k-means clustering algorithm, up to 50× energy saving can be achieved by allowing a classification-accuracy loss of 5% [46]. Indeed, due to redundancy of inner calculations, some applications are characterized by an inherent error resiliency; therefore, by relaxing functional requirements of a computing system, AxC enables to carefully trade limited quantity of accuracy off for performance, such as computation speed, throughput and, for integrated circuits, occupied silicon area.

In other words, the AxC paradigm exploits the gap between the level of accuracy required by the applications/users and that provided by the computing system, with the latter being often far lower than the former, for achieving diverse optimizations. Thus, this design paradigm has the potential to benefit a wide range of applications, including data analytic, scientific computing, multimedia and signal processing, and machine learning.

Anyway, exploiting AxC requires coping with

(i) the designation of parts of the considered software or hardware component which are suitable to be approximate;

(ii) the approach to introduce actual approximation;

(iii) the selection of appropriate error metrics, which generally depend on the particular application;

(iv) the actual error-assessment procedure, to guarantee output quality constraints are met [47], and, finally

(v) the DSE, to select the best approximate configurations among those generated by a certain approximation technique.

As for the first two of the aforementioned issues, pinpointing approximable code or data portions may require the designer to have deep insights into the application. Moreover, since a naive approximation approach – such as the uniform one – is unlikely to be efficient, and since no approach can be universally applied to all approximable applications, the approximation approach needs to be determined on a per-application basis by the designer. Approaches to identify approximable portions within applications and a survey of approximation techniques are provided in Section 2.2.1 and Section 2.2.2.

As for error assessment, it typically requires the simulation of both exact and approximate applications, nevertheless Bayesian inference [165] or machine-learning based approaches [126] have been proposed in the scientific literature. Metrics and assessment techniques will be further discussed in Section 2.2.3.1 and Section 2.2.3.

Finally, concerning DSE, initial approaches either combine multiple design objectives in a single-objective optimization problem or optimize a single parameter while keeping the others fixed. Therefore, the resulting solutions are centered around a few dominant design alternatives and do not explore the whole Pareto-front [56]. Recently published works address the circuit design problem by using MOP to search for Pareto-optimal approximate circuit implementations [156]. Unfortunately, such approaches did not focus on complex systems, rather on arithmetic components, such as adders and multipliers, since they are building-blocks for more complex designs.

## 2.2 Challenges in Approximate Computing

As we outlined before, an effective use of the AxC requires addressing several issues. Here, some contribution from scientific literature are briefly summarized below.

### 2.2.1 Identifying approximable portions within applications

Finding approximable portion of an algorithm, or data error-resilient data, is the crucial initial step in AxC. Although this is straightforward in several cases, in other it may

require the designer to have deep insights into the application being approximate or even its implementation. Error-injection is quite common as an approach to find the data or operation that can be approximated with little impact on quality of result. Nevertheless, it is not always suitable. Therefore, several approaches have been proposed in scientific research. These can be roughly classified as: (i) using directives to specify to the compiler which parts are approximate, (ii) adding support for defining approximate areas directly in the programming languages, or (iii) automatically identifying which code/data are approximable.

Although manually annotating, using appropriate directives, the code to approximate could seem a naive choice, when used in conjunction with an appropriate technique it allows achieving significant results. Authors of [145], for instance, adopts OpenMP-style directives to annotate the code at design time. Then, at run time, based on accurate or approximate directives, the floating-point units are promoted/demoted to accurate/approximate mode to match program region requirements. This allows maintaining an acceptable quality loss in error-tolerant applications, and reduces recovery overhead in error-intolerant applications, while providing significant energy savings in both types of applications. In [171] an almost identical approach is adopted for skipping tasks: depending on the impact of a task on the final output quality, a programmer can express its significance using compiler directives. The programmer can also optionally provide a low-overhead inexact version of a task. Further, the acceptable quality-loss is specified in terms of fraction of tasks to be executed precisely. Based on this, the run time system employs inexact versions of less-important tasks or drops them completely.

Instead of using directives, in [152] extends the programming language with approximate data-types; then, they use type qualifiers for specifying approximate data and separating precise and approximate portions in the program. For the variables marked with approximate qualifier, the storage, computing, and algorithm constructs used can all be approximate. This approach can be successfully adopted not only for software applications, rather on hardware too. Authors of [184], for instance, present annotations for providing suitable syntax and semantics for approximate hardware design in Verilog, allowing the designer to specify both critical (precise) and approximable portions of the design. In addition, such syntax allows reusing approximate modules in different designs while having different accuracy requirements without

requiring reimplementation. A different approach is proposed in [38]: authors propose a programming language, namely *Rely*, that allows the programmer to quantitatively specify the reliability of a program. *Rely* allows associating a measure of the reliability to variables or to the return values of functions. Then, based on these specifications, the compiler is able to determine whether or not such data can be processed and stored using approximate hardware.

Resiliency to error can be generally automatically analyzed while focusing on small portions of an application. In [47], for instance, authors consider inner loops of an application as atomic kernels. They introduce random errors into the output variables while monitoring the latter. If the output does not meet the quality criterion or if the application crashes, the kernel is marked as sensitive; otherwise, it is marked as potentially resilient. A subsequent step further explore potentially resilient kernels, in order to attempt the applicability of various approximation strategies. A similar approach has been proposed in [149]. Authors first collects the variables of the program and the range of values that they can take. Then, the values of the variables are perturbed, and the new output is measured. By comparing this against the correct output, which fulfills the acceptable quality threshold, the contribution of each variable in the program output is measured. Based on this, the variables are marked as approximable or non, approximable. Anyway, injecting error is not the only viable approach. The spatial or temporal correlation of inputs (e.g., pixels of an image or frames of a video), for instance, can be exploited to identify computations that are amenable to approximation [144].

### 2.2.2 Techniques to introduce approximation

Once portions, or data, to be approximate have been identified, being able to introduce approximation is not a straightforward matter, and may require coping with several technical challenges.

Concerning circuit approximation, the scientific literature distinguishes in timing and functional techniques [147]. The former consists of forcing the circuit to operate on reduced voltage or higher frequency than nominal ones, while the latter includes altering the logic being implemented. Technology-independent functional approximation

currently represents the most popular technique on how to introduce approximations to hardware components. In the following, we briefly discuss several proposals from the scientific literature, including working with reduced precision, skipping loop iterations, memory accesses and even tasks, performing operations using inexact hardware and so forth.

One of the most commonly adopted techniques to introduce approximation is precision-scaling, or bit-width reduction, which, essentially, reduces the amount of bits used for representing input data and intermediate operands [164]. Typically, the least significant bits are masked or neglected in order to save resource, allowing the use of smaller and faster Floating-Point Units (FPUs) for several computations. Moreover, it allows using a lookup tables for performing multiplication and add operations, or event turns a floating-point operation into a trivial one (e.g., multiplication by one), which would not require use of a dedicated FPU. In addition, precision-scaling essentially combines close values to a single value, which increases the coverage of the memoization technique [186].

The memoization approach works by storing the results of functions for later reuse with identical function/inputs. However, if reusing the results for similar functions/inputs is allowed, memoization inherently allows approximation. For instance, authors of [98] propose a value cache to implement a clumsy hardware memoization mechanism. This is able to perform partial matches, i.e., reducing the arithmetic precision of the input parameters, thus increasing significantly the volume of successful value reuses.

Load value approximation, that is another memory-based approximation approach, has been inspired by observing that on a load miss in a cache, the data must be fetched from the next-level cache or main memory, which incurs large latency. Load value approximation leverages the approximable nature of applications to estimate load values, thus allows a processor to progress without stalling for a response. This hides the cache-miss latency. Indeed, when compared to traditional load-value predictors, using load-values approximation implies that fetching a cache block on each cache miss is not required, which reduces the memory accesses significantly. In [121], authors prove that the technique provides significant speed-up and energy saving with negligible degradation in output quality when adopted for graphics applications. In [185] the technique is adopted for alleviating the bandwidth bottleneck in General Purpose -

Graphic Processing Unit (GP-GPU) systems. Loads that do not deal with memory accesses and control flow are identified, and ones that cause the largest fraction of misses are selected as candidate for approximation. By individually approximating each of these loads, their impact on quality is measured and the loads leading to smaller degradation than a threshold are selected for approximation. When these loads miss in the cache, the requested values are predicted; however, no check for mispredictions or recovery is performed, which avoids pipeline flush overheads. In order to deal with Single-Instruction-Multiple-Data (SIMD) and Single-Instruction-Multiple-Thread (SIMT), which require predicting values for multiple threads, authors leverage the value similarity across accesses in adjacent threads to design a multi-value predictor, which significantly reduces the overhead of predicting values separately.

Loop-perforation, that works by skipping some iteration of a loop to reduce the computational overhead, is also quite a widespread approach. The technique has proven to be effective when applied to several computational patterns, such as the Monte Carlo simulation, iterative refinement, and search space enumeration [160]. Furthermore, it is also suitable on memory-accesses and tasks, as experimentally proven in [151, 71].

The usage of custom approximate hardware to implement basic-blocks of high-level applications is another quite common technique. Intentionally-approximate elementary circuits, including different implementations of adders [74, 182, 11] and multipliers [41, 115], have been proposed in the scientific literature since the early days of AxC. Moreover, recently many libraries consisting of thousands of elementary approximate circuits have been proposed in the scientific literature, supplying hundreds of implementations of even a single arithmetic operation [127, 94].

In [13], a framework relying on inexact computing to perform the DCT computation for the JPEG has been proposed. The framework acts on three levels: (i) at the application level, it exploits human insensitivity to high-frequency variation to use a filter and discard high-frequency components; (ii) at the algorithmic level, multiplier-less fast algorithms are employed for the actual DCT computation on integer coefficients; (iii) at hardware level, rather than using a simple truncation for adder circuits, authors used inexact adder cells to compute the less significant bits instead of the classic full-adder cell. Although the topic is investigating the combined effects of these levels on the final JPEG, the approach paves the way for building a complex accelerator from smaller and simpler approximate components.

In [126], circuits from a library of approximate components are selected to generate an approximate accelerator for a given application. On the basis of contributions from single components, machine learning techniques are adopted to estimate the overall quality and hardware cost of the accelerator, without requiring simulations and synthesis. A similar approach has been presented in [39]. A set of analytical models of quality and resource requirements are derived for a library of approximate components. Then these are used to estimate resource needed and accuracy of accelerator designed through high-level synthesis of C language description.

### 2.2.3 Assessing error due to approximation

Output monitoring is mandatory to ensure quality constraints are met, and often this is the most time-consuming phase of an approximation approach, since typically it requires the simulation of the whole approximate application on a significant data set [169].

Simulation-based estimation, in facts, is undoubtedly the simplest and, albeit naive, it is one of the most commonly adopted approach for error estimation. This naive approach consists of running the approximate application several times while comparing its outcomes with the non-approximate application. The comparison is achieved through the adoption of an appropriate error metric, and if the accuracy of the approximate application is not satisfactory, that given approximate configuration is discarded. The simplicity of the technique is also its main drawback: the general approach has often to rely on large and computation intensive campaigns of executions and profiling of the target application; therefore, every time a new approximate configuration is considered, the application must be executed and the corresponding metric computed again. Since the above process iterates until a desired level of accuracy is reached, the cost depends on the final amount of runs to reach it. Besides low scalability, the simulation-based approach lacks of strong guarantees when simulating the application for a random subset of the possible inputs only.

An interesting approach to overcome the highlighted issues is discussed in [138]. The authors present a framework for analytically estimating the output quality of common Digital Signal Processing (DSP) blocks that utilize approximate adders. The

error is considered as an additive white noise, and its impact on several DSP blocks – including Finite Impulse Response (FIR) filters, DCT and Fast Fourier Transform (FFT) – is estimated using a mathematical model. The latter is also compared to simulation results, by using the Signal-to-Noise Ratio (SNR) as metric, exhibiting, on average, 2.5 dB inaccuracy. Nevertheless, the white noise assumption is not always correct, since the error is dependent on the input [119].

Authors of [165, 166] proposes a stochastic approach based on a Bayesian prediction model able to estimate the error affecting the result of a complex application when precision-scaling is applied to different portions of the computation. The application is modeled in the form of a Bayesian network whose nodes represent data and operators, while arcs model the data-flow. Overall, the network models the error propagation along the data-flow of the application. To construct the model, the application is profiled only once to extract its data-flow. Then, the operators are characterized with required probabilities to be embedded in the model. Finally, the error distribution of the application can be quickly estimated exploiting the Bayesian inference theory.

Recently, machine-learning based estimation approaches have spread. Authors of [126], for instance, exploit computational models developed using machine-learning to predict both the induced error and savings due to approximation. The models are constructed independently, using a suitable supervised machine-learning algorithm. The learning process is based on providing example input-output pairs, i.e., a particular approximate configuration. The input is represented by a vector containing a subset of hardware and quality parameters for each of the approximate sub-parts realizing one of the operations as defined by the configuration. The output is a single value of either quality or result or hardware cost that is obtained by simulation and synthesis of the concrete accelerator with the given configuration. For learning, the authors have to generate a training and a validation set, typically containing from hundreds or thousands of configurations. Moreover, since the models are used for determining a relation between two different configurations, authors do not focus on their accuracy, rather their fidelity. The latter tells how often the estimated values are in the same relation (<, = or >) as the real values for each pair of configurations.

Probabilistic techniques, those based on Bayesian inference, and those based on machine-learning are able, however, to provide only an estimate of the error and, as with non-exhaustive simulations, offer no guarantees as to the maximum value that the

error can yield. Hence, various analytical and formal approaches have been proposed and applied for exact quantification of the error. Furthermore, they do not make any assumption on the structure of the approximate circuits, and they can be applied to determine almost every error metric [169].

Authors of [172, 146], for instance, use auxiliary circuits, called *miters*, to quantify the approximation error using formal verification techniques. Miters combine the original circuit and the approximate circuit and, in order to check whether a predefined worst-case error is violated by the candidate approximate circuit, a pseudo-Boolean SATisfiability (SAT) solver and Integer Linear Programming (ILP) was employed. Nevertheless, ILP does not scale with circuit complexity. Therefore, formal methods for error assessment in approximate digital circuits are nowadays based on SAT solvers and Binary Decision Diagrams (BDDs) [41].

#### 2.2.3.1 Metrics for error assessment

As mentioned, quality metrics bound the type and amount of error that can be introduced during approximation; therefore, they generally depend on the particular application being considered.

However, in essence, all of these metrics seek to compare some form of output (depending on the application, e.g., pixel values, body position, classification decision, execution time) in the approximate computation with that in the exact computation.

For general logic, where no additional knowledge is available and where there is not a well-accepted error model, Hamming distance or error-rate are typically employed. In some cases, neither the Hamming distance nor the arithmetic metrics provide a satisfactory assessment of the quality of approximate circuits. Hence, various problem specific error metrics have been introduced.

Therefore, quality metrics are generally classified w.r.t. the application domain, distinguishing into non-domain-dependent metrics and domain-dependent metrics. Examples of domain-dependent metrics are, clustering accuracy and mean centroid-distance, which are common metrics for k-means clustering, or classification-accuracy for machine-learning predictors.

Non-domain-dependent metrics are broadly classified in three categories, i.e.: (i) metrics constraining error magnitude, (ii) metrics bounding error frequency, and (iii) composite metrics constraining both error magnitude and frequency For the sake of brevity, this manuscript just discusses the most commonly adopted domain-independent metrics, although a complete survey is available in [147].

#### 2.2.3.1.1 Metrics constraining error magnitude

**The Absolute Worst-Case Error (AWCE)** It is defined in Equation (2.1), constrains the absolute difference in magnitude between the outputs of the original and approximate circuits to be less than a specific threshold for each input.

$$e_{awce}(f, \hat{f}) = max_{\forall x \in \mathbb{B}^n} \left| f(x) - \hat{f}(x) \right| \tag{2.1}$$

$f(x)$ and $\hat{f}(x)$in Equation (2.1) are the output of the original (exact) application and its approximate counterpart, respectively.

**The Relative Error Magnitude (REM)** It is defined in Equation (2.2), and it bounds the absolute value of the difference between 1 and the ratio of the approximate output to the original output by at most a certain margin for every input.

$$e_{rem}(f, \hat{f}) = max_{\forall x \in \mathbb{B}^n} \left| 1 - \frac{\hat{f}(x)}{f(x)} \right| \tag{2.2}$$

**The Average Error Magnitude (AEM)** It is defined in Equation (2.3). It bounds the absolute difference in magnitude between the approximate output and the original output, averaged over all the $n$ possible circuit inputs.

$$e_{prob}(f, \hat{f}) = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} \left| f(x) - \hat{f}(x) \right| \tag{2.3}$$

**The Mean Squared Error (MSE)** It is defined in Equation (2.4), and it bounds the mean of the squared difference between the original and the approximate output across all possible inputs to be less than a given threshold.

$$e_{prob}(f, \hat{f}) = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} \left( f(x) - \hat{f}(x) \right)^2 \tag{2.4}$$

#### 2.2.3.1.2 Metrics bounding the error frequency

**The Error Probability** It is defined as the fraction of input vectors for which the approximate circuit output differs from the original circuit output, as in Equation (2.5).

$$e_{prob}(f, \hat{f}) = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} [\![ f(x) \neq \hat{f}(x) ]\!] \tag{2.5}$$

In Equation (2.5), the $[\![\cdot]\!]$ notation denotes the Iverson bracket, i.e.

$$[\![P]\!] = \begin{cases} 1 & i.f.f.\ P\ is\ true \\ 0 & otherwise \end{cases} \tag{2.6}$$

with $P$ being a preposition.

**The Bit Error Probability** It is a slight variant of the error probability metric: the error probabilities of individual output bits are bounded separately, as defined in Equation (2.7).

$$e_{prob}(f, \hat{f})_i = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} [\![ f(x)_i \neq \hat{f}(x)_i ]\!] \tag{2.7}$$

## 2.3 Design methodologies targeting hardware applications

Circuit approximation can be undoubtedly performed manually. However, the current trend is to develop fully automated functional approximation methods that can be integrated into computer-aided design tools for digital circuits. The goal is to obtain a tool performing automatic approximation of digital circuits independent of their structure. In this Section, we review systematic methodologies targeting both combinational and sequential circuits from the scientific literature, highlighting their main features, and the innovative contributions brought to the scientific literature.

### 2.3.1 Achieving power-savings through careful data sizing

One of the first attempt to define a systematic methodology for digital circuits approximation is [135]. The methodology focuses on minimizing power consumption by making use of the precision-scaling technique. It needs a C/C++ model of the design to be approximated, a set of error constraints on output variables, and a set of variation ranges for input variables or, alternatively, an input dataset.

The workflow includes a first static analysis of the given model, during which arithmetic operations are performed on ranges, instead of single values. This allows to estimate the variability range of intermediate and output variables, and to determine the optimal number of fractional bits required for their representation. The latter is exploited to perform a floating-to-fixed point conversion. A further dynamic analysis, which includes range analysis, automatic differentiation and branch analysis, is, then, performed. The first one allows to determine whether the floating-point representation may be more effective than the fixed-point one when dealing with very small values, and to detect input patterns, which can be exploited for variable-to-constant conversions. Automatic differentiation is adopted to cope with complex functions, such as trigonometric ones. These make the error analysis cumbersome, since they are typically implemented by specific IP-cores, whose actual implementation may be not known during the analysis. Finally, a branch analysis is performed, by splitting the design in basic blocks, which are ranked based on their execution frequency. This allows

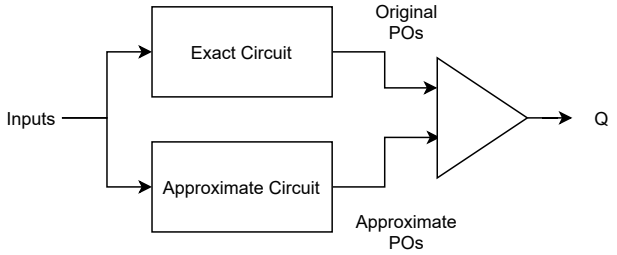



Figure 2.1: Quality Constraint Circuit from [172]

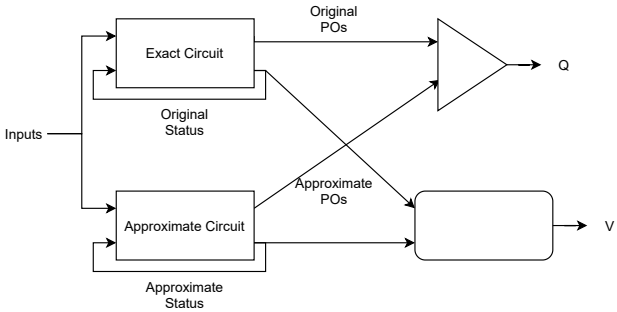



Figure 2.2: Sequential Quality Constraint Circuit from [146]

precision of variables along less frequently executed blocks to be further reduced.

### 2.3.2 Exploiting Boolean algebra towards functional approximation

Instead of a C/C++ model, SALSA [172] consider circuit implementations at the Register-Transfer Level (RTL). SALSA encodes both the type and the amount of approximation allowed by the considered application in one, or more Boolean logic functions, defining the Quality Constraint Circuit (QCC), which is depicted in Figure 2.1. The QCC consists of three blocks: a structural description of the circuit to be approximate, the approximate circuit and the quality function, that defines error constraints to be satisfied.1 During the synthesis process, the approximate circuit is iteratively evolved while preserving the $Q = 1$ invariant. The approximation is introduced by exploiting the Observability Don't Care (ODC) set of nodes and External Don't Care (ExDC) set

of Primary Outputs (POs). As for the former, it is the set of input values for which POs are insensitive to the output of the considered node. As for the latter, it is the set of Primary Input (PI) combinations for which that PO is a don't care. Hence, the ODC set for POs of the approximate circuit are firstly identified; then, they are considered as ExDCs for that POs, in order to simplify the cone of logic generating that output using standard don't care based synthesis techniques.

The methodology implemented in SALSA, which was originally intended to work only on combinational circuits, has been extended to sequential circuits by authors of [146]. Working on sequential circuits require addressing several challenges, the estimation of the impact of approximation on the output quality, observed after multiple cycle of operations. Therefore, authors of [146] exploits the Sequential Quality Constraint Circuit (SQCC), that is depicted in Figure 2.2, to characterize the impact of approximation. As its combinational counterpart, the SQCC consists of three components, i.e., the original sequential circuit, the approximate circuit, and the Quality Evaluation Circuit (QEC). The latter encodes the quality constraints to be met while monitoring both POs and status registers, in order to compute the $Q$ and the $V$ bits. These, respectively, are set when quality constraints are satisfied, and when operations performed by the approximate circuit has been completed, therefore its POs are ready to be evaluated.

Combinational blocks within the considered circuit are identified first; then, a gradient-descent heuristic searches for their optimal quality-energy operating point, by taking into account the (i) proportion of energy required by the block w.r.t. the whole circuit, (ii) the energy saving obtained by approximating the block and (iii) the error introduced due to the approximation. The best configuration is then selected, and the quality constraints are checked using a sequential quality constraint circuit. The process is repeated until no block can be further approximated without violating constraints.

In order to guarantee output quality, formal verification is performed leveraging the following properties: (i) safety: in all possible states of the SQCC, if $V$ is true then $Q$ should be true, and (ii) liveness: $V$ eventually becomes true along all possible paths through the space state of the SQCC. The former property ensures that whenever both the original sequential circuit and the approximate one have produced their outputs, i.e., $V$ is high, the approximate circuit must satisfy the quality constraints encoded by

$Q$, i.e., $Q$ must be is high. The second property states that both the original and the approximate circuits eventually produce their respective outputs.

The process is performed by making use of the time expansion-technique, which means that the SQCC is iteratively unrolled until the $V$ signal is high; then, quality constraints are checked.

In [79], Boolean Matrix Factorization (BMF) is exploited to factorize truth tables of multi-outputs digital circuits. Given a multi-output Boolean logic function, its truth-table is factorized while using the factorization factor $f$, in order to produce two sub-matrices, namely $B$ and $C$, which correspond to the truth-table of a compressor and a decompressor circuit. This allows to force any arbitrary circuit to compress as much information as possible in $f$ intermediate signals. Hence, such information can be decompressed using the decompression circuit. Approximation can be introduced by not preserving the equality between the starting truth-table $M$ and the product of the matrices $B$ and $C$.

Since BMF is NP-hard, and since its complexity grows exponentially with the size of the matrix to be factorized, the truth-table is split in a number of smaller tables, each factorized in isolation. Nevertheless, decomposing a circuit in smaller sub-circuits does not mean the latter can be tested for error in isolation for error assessment; thus, the circuit is recomposed, and error assessment is performed using Monte Carlo simulations.

### 2.3.3 Approximate circuit by means of evolvable hardware

Authors of [156] pointed out that the circuit design problem can be formulated as a MOP, and solved by means of a search algorithm. In facts, according to authors, when compared with other works, a MOP-based approach provides many alternative solutions showing high-quality trade-offs between key design objectives. This originates in [56]. There, the authors prove that combining multiple design objectives in a single-objective optimization problem results in solutions centered around a few dominant design alternatives, that, hence, cover only a portion of the whole Pareto-front.

The approach in [156] is a Cartesian Genetic Programming (CPG)-based general-

purpose method for automated functional approximation of digital circuits at gate or RTL level. Every candidate circuit is represented as a special two-dimensional grid consisting of $N = n_c \times n_r$ nodes, with $n_c$ and $n_r$ being columns and rows of the grid, respectively. The amount of primary inputs and primary outputs of the circuit are $n_i$ and $n_o$, while each component has up to $n_a$ inputs and $n_b$ outputs. The type of components on the grid depends on the chosen abstraction level.

In order to enable circuit topology specification, each component is assigned with a unique index and, in the CPG context, it is represented using $n_a + 1$ integers. The former specify the destination indexes for the component's inputs, while the latter specifies the logic function being implemented. Hence, the whole circuit is represented using $N_g = n_c \times n_r \times (n_a + 1) + n_o$ integers, i.e., genes, which compose a chromosome.

Every chromosome is, in the CPG context, a point in a multidimensional design space. New designs are created introducing mutations, i.e. random modifications which can affect either input connections, the implemented function or the component output connection. The $1 + \lambda$ generation strategy is adopted and, in order to converge to the Pareto-front, the NSGA-II search algorithm is adopted. The approach is evaluated using several arithmetic circuits as case studies, optimizing for error, circuit delay and power consumption.

### 2.3.4 Building approximate circuits from library of approximate components

Recently, many libraries consisting of thousands of elementary approximate circuits have been proposed in the scientific literature, supplying hundreds of implementations of even a single arithmetic operation [127, 94].

Authors of [126] addressed how to effectively combine circuits from libraries to design complex approximate accelerators. Their proposed methodology aims at providing designers with a set of Pareto-optimal configurations where the quality of results and hardware costs are both optimized. It requires a hardware description of the accelerator to be approximate, the corresponding software model and benchmark data, and it is structured on three steps: (i) the library of approximate components is preprocessed, se-

lecting a suitable set of approximate circuits and discarding irrelevant ones, on the basis of their quality w.r.t. a given application and their hardware cost; (ii) machine-learning algorithms are employed to build predictors, enabling quality of result and hardware cost estimation requiring neither simulations nor hardware synthesis of approximate configurations; (iii) the Pareto-front reflecting quality and hardware cost is iteratively constructed using a two-phases approach: first, predictors built in the preceding step are employed to quickly build a pseudo-Pareto-front, which is then refined towards the final Pareto-front using quality and hardware estimation provided by simulations and hardware syntheses of approximate configurations. During the library pre-processing step, the target accelerator is profiled, computing the Weighted Mean Error Distance (WMED), according to Equation (2.8), for each operation $M_k$ within the Data-Flow Graph (DFG) of the accelerator, where $I$ is the set of inputs from the benchmark data and $P_k(i)$ is the probability with which a vector $i \in I$ can constitute the input of the operation $M_k$. Then, approximate circuits providing Pareto-optimal trade-offs between hardware costs and WMED are selected for the next step.

$$WMED_k(\tilde{M}) = \sum_{\forall i \in I} P_k(i) \times |M_k(i) - \tilde{M}_k(i)| \tag{2.8}$$

The WMED, area, power and delay of all the selected circuits are, then, considered to build predictors for quality of results and hardware cost estimation. Several learning algorithms and models are compared, with regression algorithm and random forest providing the best performance in terms of accuracy and fidelity. During the DSE, the iterative heuristic, which is basically a hill-climbing, starts from a set of randomly generated candidate solutions. At each iteration, a candidate solution is selected and randomly altered: children solutions are derived from parents by picking-up a random circuit from the library. Then, quality and hardware costs are estimated using predictors, and whether a child dominate its parent it is selected for further iterations, becoming part of the archived candidate solutions. As it is easy to grasp, the quality of the so build Pareto-front strictly depends on the fidelity of predictor models. Therefore, during the second phase of the DSE, the pseudo-Pareto-front is refined using actual simulations and hardware syntheses.

# Chapter 3

# Application-driven, Multi-Objective Approximate Design Methodology

This Chapter discusses our AxC and MOP-based design methodology. Specifically, we will describe main steps of the method that we propose, and how the method helps in addressing the challenges that the AxC paradigm poses to the designer. We also devote particular emphasis to the steps of the methodology that can be automated, including the generation of approximate variants for a give application, and the selection of optimal trade-offs between quality of results and hardware-requirements through the use of a MOP-based DSE.

## 3.1 Application-driven design approach

As mentioned in the previous Chapter, the AxC design paradigm emerged as one of the most promising breakthroughs to overcome design challenges posed by the design of modern computing systems, including the slowdown in growth in performance gains and the increase in energy consumption. Indeed, exploiting the redundancy of data or inner computations, or even end-user perceptual limitations, allows relaxing functional requirements of computing systems, and enables trading limited quantity of accuracy off for performance, such as computation speed, throughput and, for integrated circuits, occupied silicon area.

Image, video and audio processing applications, for instance, are highly error-tolerant since human senses cannot often perceive degradation in performance, such as quality of visual and audio information. Machine-learning applications, including classification, recognition, mining, and synthesis, exhibit a high level of error resilience, meaning that such applications are able to produce acceptable outputs despite some underlying computations being incorrect or approximate. The inherent resilience of these applications can be attributed to several factors, including: (i) the significant redundancy which is present in large real-world data sets that these applications process; (ii) the computational patterns they exploit, such as statistical aggregation, majority voting and iterative refinement, that intrinsically attenuate or correct errors due to approximations, and (iii) outputs equivalency, i.e., no unique golden output exists, or small deviations in the output cannot be perceived by users.

This error-resiliency can be effectively exploited by using the AxC design paradigm, relaxing the traditional requirement of exact – i.e., numerical or Boolean – equivalence between the specification and implementation, and allowing applications to produce outputs of acceptable quality, rather than “correct” output.

Anyway, as discussed in Section 2.1, there are several challenges to cope with in order to effectively exploit the AxC design paradigm, and though diverse research works in the scientific literature proposed well-founded approaches addressing the above-mentioned challenges, there are still plenty of open ones holding AxC back from wider employment. In particular, the key point is the lack of a general and automatic DSE methodology. Indeed, existing AxC design tools consider specific

transformations and specific domains, and they are not fully automatic, providing only a guided approach for approximation.

Conversely, in the following we discuss a generic, MOP-based and fully automatic method to design hardware accelerators for error-resilient applications.

When presenting our methodology, we discuss the phases it breaks into, including (i) how to identify which part of the application is amenable for approximation and (ii) a suitable approximation technique, (iii) how MOP-based DSE can be defined, and, finally, (iv) hot to pinpoint suitable fitness-functions for error assessment and savings estimation in order to effectively drive the DSE toward Pareto-optimal approximate configurations.

### 3.1.1 Identifying approximable portions and suitable approximation techniques

The first challenge to be addressed when dealing with the AxC is identifying error-resilient – i.e., approximable – data or portions of a given algorithm/application, and, consequently, a suitable approximation technique. Although it seems trivial, this step of the methodology is rather quite crucial. Indeed, as we discuss in the following, an improper design-choice concerning either parts to be approximate, or the technique to be adopted, impacts all the subsequent phases.

Despite many of the methods from the scientific literature claim to be generic, they actually require the designer to have deep knowledge of the target application in order to choose a suitable approximation technique. Unfortunately, this may be cumbersome, or even not possible: there are plenty of applications for which, albeit conceptually simple, having deep understanding is very difficult indeed, e.g, DNNs and DT MCSs. Furthermore, once portions to be approximate have been correctly pinpointed, and a suitable approximate technique selected, the latter have to be applied to the former in order to introduce approximation. However, manual introduction of approximation within applications is definitely inconvenient, due to their complexity or due to the amount of data/operations amenable for approximation.

Conversely, on one hand, our methodology requires only minimal knowledge of the target application and, on the other hand, it provides the designer with a systematic approach to automatically generate approximate variants of a given application, i.e., implementations in which approximable parts are superseded by making use of approximate components.

In general, given an algorithm implementation, in order to automatically generate approximate variants while having control on the error, it is needed to collect information on the operations suitable for approximation. The gathering process can be automated by analyzing the AST of the application, while AST manipulation exploiting *mutators* [22] allow the automatic generation of approximate variants.

Mutators are defined as a set of rules to search and modify the AST; the rule definition is generally application-independent, and does not require the designer to know the algorithm or its specific implementation. Furthermore, mutators do not depend on the specific approximation technique being adopted, and they allow to effectively introduce a suitable tuning knobs for approximation, replacing exact operations within the AST using their approximate counterparts. Consider, for instance, an approximate multiplier designed using the precision-scaling technique, and let the Number of Approximate Bit (NAB) be the parameter for such approximation and suppose the approximate operation truncates the least $nab$ significant bits of operands, with $nab$ being configurable. Setting a value for the $nab$ parameters tunes the approximation degree, resulting in an *approximate configuration* of the algorithm. Mutators can also be exploited to implement the inexact-component technique. Indeed, exact multiplications can be automatically replaced using a mutator that allows selecting which implementation to be adopted among those provided by a given library, e.g., the EvoApproxLib library [127]. In this case, the configuration parameter would allow selection of an optimal multiplier implementation with respect to a given error metric, required silicon area, and power dissipation.

### 3.1.2 Optimization and design-space exploration

The number of approximate variants and, consequently, the number of approximate configurations, grow quickly with the number of parts suitable for approximation. Con-

sider, for instance, an algorithm implementation with $n$ approximable operations, each allowing $k$ different degrees of approximation: $\binom{n}{j}$ different approximate variants can be defined by simultaneously approximating $j$ operations, and $k^j$ different approximate configurations can be defined for each of the variants. Therefore, the total number of approximate configurations is $\sum_{i=1}^{n} k^i \times \binom{n}{i}$. At this point, the main challenge is to find values for the approximation parameters leading to the Pareto-optimal trade-offs between performance gains and accuracy losses.

In facts, each one of the introduced approximation parameters impact both accuracy and savings. Hence, the automated design of approximate circuits is inherently a MOP in which a circuit satisfying user-defined constraints and showing the desired trade-off between the quality and other electrical parameters is sought in the space of all possible implementations [170]. As we mentioned, most of the approximation approaches either combine multiple design objectives in a single-objective optimization problem or optimize a single parameter while keeping the others fixed. Therefore, the resulting solutions are centered around a few dominant design alternatives and do not explore the whole Pareto-front [56]. We propose to find Pareto optimal configurations for approximation parameters through an automatic MOP-based DSE, which is not only tailored to the target application, yet it considers the latter target as a whole. Indeed, recent works addressing the circuit design problem as MOP, e.g., [156], did not focus on complex systems, rather on arithmetic components, such as adders and multipliers, since they are building-blocks for more complex designs.

In the following we provide the reader with the required knowledge concerning MOP.

#### 3.1.2.1 Multi-objective Optimization Problems

Basically, a MOP consists of a set of *fitness-functions* to minimize/maximize at the same time and a set of *constraints* to be met, as reported in (3.1),

$$\begin{aligned} &\Gamma = \{\gamma_i : A \to \mathbb{R},\ i = 1 \cdots k\} \\ &\Psi = \{\psi_j : A \to \{0,1\},\ j = 1 \cdots l\ \} \\ &A \subseteq \mathbb{R}^n \end{aligned} \tag{3.1}$$

where $\Gamma$ and $\Psi$ are the set of fitness-functions and the set of constraint-functions, respectively. While the functions of the former set assume values in $\mathbb{R}$, or its subset, the constraint functions assume either the value 1 or 0 to indicate that the constraint is or is not met, respectively. Equation (3.2) describes the set of solutions for (3.1). For non-trivial MOPs, $|X| > 1$, where $|\cdot|$ expresses the size of the set, i.e., the number of elements it contains.

$$X = \{x \in A : \gamma_i(x) \leq \gamma_i(x'),\ x \vdash \psi_j\ x' \neq x,\ i \in [1,k]\,,\ j \in [1,l]\} \tag{3.2}$$

Indeed, since different objectives (i.e., fitness-functions) often represent conflicting goals, the DSE goal is to seek for a set of equally good solutions being close to the *Pareto-front* (3.2). Let us consider two solutions, $x, y \in X : x \neq y$, $x$ is said to *dominate* $y$ i.f.f. (3.3) holds, i.e., $x$ shows better or equally good objective values than $y$ in all objectives and at least better in one objective. If a solution is not dominated by any others, it is called a *Pareto-optimal* solution.

$$x \prec y \iff \gamma_i\,(x) \leq \gamma_i\,(y)\,\forall i \in [1,k]\ \wedge\ \exists j \in [1,k] : \gamma_j\,(x) < \gamma_j\,(y) \tag{3.3}$$

Due to the rapid growth of the size of the solution space as the number of decision variables, fitness-functions and constraints increases, using exact solving algorithms turns out to be very computation-intensive. Consequently, a variety of heuristics aiming at producing an approximation of the Pareto-front have been proposed in the scientific literature. Two of the most commonly adopted ones are the Genetic Algorithm (GA) and the Archived Multi-Objective Simulated Annealing (AMOSA), which will be discussed in Section 3.1.2.2 and Section 3.1.2.3, respectively.

#### 3.1.2.2 The Genetic Algorithm heuristic

GAs [120], a subclass of Evolutionary Algorithms (EAs), have been largely used in the literature to find Pareto-fronts for MOPs: they are inspired by and also borrows terminology from the evolutionary theory. The evolution process starts from an *initial population*, which is typically randomly generated or seeded in areas where optimal solutions are likely to be found.

One of the most relevant concept behind GA is the *phenotype-to-genotype mapping*,

that, for each member of the population being evolved, encodes observable features, i.e., the element's phenotype, through the use of a suitable representation. The latter, which represents the element's genotype, is commonly referred to as *chromosome*, and consists of a set of parameters defining a candidate solution to the problem that theGA is trying to solve. Borrowing terminology from biology, elements of a chromosome are called *genes*, and each element of a population is called an *individual*.

For what pertains to GA, the design of the chromosome is, by necessity, specific to the problem to be solved, but, traditionally, chromosomes are represented in binary as strings of 0s and 1s; however, other encoding are also possible, and almost any representation which allows individuals to be represented as a finite-length vectors can be adopted. Nevertheless, finding a suitable representation of the problem domain for a chromosome is of major concern, since a good representation will make the search easier by limiting the search space. Conversely, a poorer representation will result in a larger, or even unfeasible, search space.

The initial population is evolved until either it converges to a set of non-dominated solutions, or a stop-criterion is reached. At each generation, fitness-functions of the MOP being solved are evaluated for each individual, and the more fit ones are selected from the current population, and further evolved to form a new population, which is used in the next generation. During the evolution process, a new *offspring* is generated through *mutation* and *crossover*. The former is a genetic operator used to maintain genetic diversity from one generation of a population to the next. It is analogous to biological mutation: it alters one or more gene values in a chromosome from its initial state. The latter, also called *recombination*, is a genetic operator used to combine the genetic information of two parents in a new offspring.

One of the most commonly adopted implementations of such operators are the simulated binary crossover and polynomial mutation [61]. The procedure of computing the offspring $x_i^{(1,t+1)}$ and $x_i^{(2,t+1)}$ from the parent solutions $x_i^{(1,t)}$ and $x_i^{(2,t)}$ involves a spread factor $\beta_i$, which is defined as the ratio of the absolute difference in offspring values to that of the parents, a random number $u_i$, a specified probability-distribution function, and an ordinate $\beta_{qi}$. The latter has to be found so that the area under the probability curve from zero to $\beta_{qi}$ is equal to the chosen random number $u_i$.

The probability distribution used to create an offspring is given by Equation (3.4),

where $\eta_c$ is any non-negative real number and $\beta_i$ is given by Equation (3.5). After obtaining $\beta_{qi}$ from the above equations, the offsprings are derived from Equation (3.6).

$$P(\beta_i) = \begin{cases} \frac{(\eta_c+1)\beta_i^{\eta_c}}{2} & if\ \beta_i \leq 1 \\ \frac{\eta_c+1}{2\beta_i^{\eta_c+2}} & if\ \beta_i > 1 \end{cases} \tag{3.4}$$

$$\beta_i = \left| \frac{x_i^{(2,t+1)} - x_i^{(2,t+1)}}{x_i^{(2,t)} - x_i^{(2,t)}} \right| \tag{3.5}$$

$$\begin{aligned} x_i^{(1,t+1)} &= \frac{(1+\beta_{qi})x_i^{(1,t)} + (1-\beta_{qi})x_i^{(2,t)}}{2} \\ x_i^{(2,t+1)} &= \frac{(1-\beta_{qi})x_i^{(1,t)} + (1+\beta_{qi})x_i^{(2,t)}}{2} \end{aligned} \tag{3.6}$$

For what pertains to the polynomial mutation operator, which is adopted to perturb offsprings, it alters a variable – i.e., a gene within a chromosome representing an individual – by following the same probability distribution as the simulated binary crossover discussed above.

**3.1.2.2.1 The Non-dominated Sorting Genetic Algorithm - II (NSGA-II)** Initial implementations of GA select the more fit individuals by making use of binary tournament selection. Nevertheless, this strategy has $O(MN^3)$ computational complexity [60], with $M$ being the number of fitness-functions and $N$ the population size. Nowadays, the most widely employed selection strategy is the NSGA-II [60], which reduces the computational complexity to $O(MN^2)$.

At the beginning, the NSGA-II builds a random initial population $P_0$ of size $N$, and sorts the latter using *non-domination sorting*, which will be presented below. Thus, each individual is assigned a *rank* equal to its non-domination level. At the first iteration, usual crossover, mutation and binary-tournament selection operator are used to build the $Q_0$ offspring population. The behavior of the algorithm during a generic i-th iteration is depicted in Figure 3.1:

(i) a combined population $R_i = P_i \cup Q_i$ is formed and sorted on the basis of non-domination;

(ii) individuals from high-rank subsets are preserved in the next iteration;

(iii) in order to select exactly $N$ individuals from the current population, individuals from the last subset are sorted on the basis of the *crowding distance*, and selected using the *crowded selection operator*.

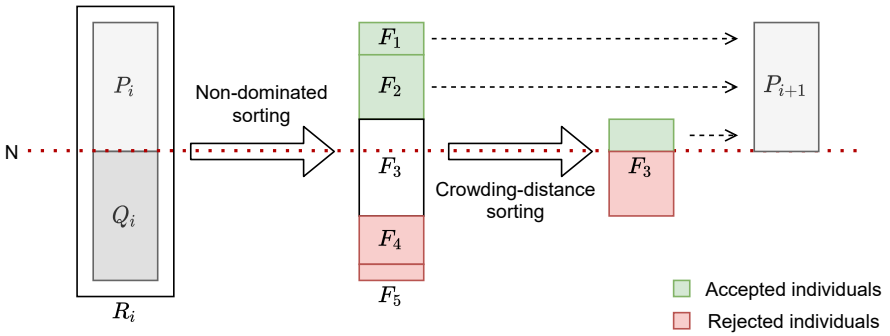



Figure 3.1: The NSGA-II selection strategy.

In order to perform non-domination sorting, for each individual $p \in P$, the NSGA-II computes

(i) the non-domination count $n_p = |D_p|,\ D_p = \{q \in P : q \prec p\}$, i.e., the number of individuals $q$ which dominate $p$, and

(ii) the set $S_p = \{q \in P : p \prec q\}$, i.e., the set of individuals $q$ which are dominated by $p$.

Individuals in the highest rank subset of $R_i$, i.e. $F_1$, have $n_p = 0$. Then, for each $p : n_p = 0$, each $q \in S_p$ is visited and $n_q$ is decremented by one. If $n_q$ becomes zero, $q$ is placed in the second-highest rank subset of $R_i$, i.e. $F_2$. The procedure is iterated until each subset is fully identified.

Along with convergence to the Pareto-front, it is also desired that an EA maintains a good spread in the obtained set of solutions. In order to preserve diversity, the NSGA-II adopts the crowding distance as a metric for the density of solutions surrounding a

particular solution. Such distance is the average distance of two points on either side of the considered solution, along each of the fitness-functions. The computation requires sorting the population $M$ times, according to each fitness-function, in ascending order. Each time the population is sorted, the boundary solutions – i.e. solutions with the smallest and the largest fitness – are assigned an infinite distance, while all other solutions are assigned with a crowding distance equal to the absolute normalized difference of fitness of the two adjacent solutions. The overall crowding distance is the sum of individual distances corresponding to each fitness-function. Using crowded distance, the usual definition of Pareto dominance (3.3) is slightly modified as in (3.7), where $\approx$ denotes $x$ and $y$ do not dominate each other and $d(\cdot)$ is the crowding distance of a given solution. The crowded distance operator preserves diversity by promoting solutions located in less crowded areas of the solution space.

$$x \prec_n y \iff x \prec y \cup (x \approx y \wedge d(x) > d(y)) \tag{3.7}$$

#### 3.1.2.3 The Archived Multi-objective Simulated Annealing heuristic

The Simulated Annealing (SA) heuristic [101] is based on the physical metaphor of the annealing process of metal and glass, mimicking the behavior of atoms when the "matter" is heated close to its melting point, in order to obtain minimal costs solutions to large optimization problems, by minimizing energy associated to a "matter" configuration. In the annealing process, materials have to be first heated close to their melting point, then the temperature is slowly lowered, and a long time is spent at temperatures near the hardening point. This yields stable low-energy states.

Since the search-from-a-point approach, there have been only a few attempts aiming at multi-objective SA. In most of the early attempts, multi-objective were achieved by combining several single-objective fitness-functions in a single fitness-function, using weighted sum. Nevertheless, results from such an approach are centered around a few dominant design alternatives and do not explore the whole Pareto-front [56].

The AMOSA [19] searching algorithm emerged as quite a promising approach, since the acceptance criterion between the current Pareto-front estimation and new solutions is based on the amount of domination, rather than plain Pareto-dominance. Moreover, it incorporates the concept of *archive*, where all non-dominated solutions

found so far are stored. Although the size of the Pareto-front is theoretically infinite, the size of the archive – i.e., the amount of diverse solutions it contains – is limited since, on one hand, the ultimate purpose of heuristics for solving MOPs is to provide the user with a set of well-distributed solutions to choose from, and, on the other hand, too large archive may negatively affect performances.

The algorithm needs the following parameters to be set a-priori:

- $HL$: the maximum size of the archive on termination, i.e., the size of the final Pareto-front estimation;
- $SL$: the soft-limit, i.e., the maximum size to which the archive may be filled before clustering is used to reduce its size to $HL$;
- $T_{max}$: the initial temperature;
- $T_{min}$: the final temperature;
- $\alpha$; the cooling factor;
- $I$: the number of iterations at each temperature.

The first step of the AMOSA algorithm is the archive building: $\gamma \times SL,\ \gamma > 1$ initial solutions are generated, starting either from random points or from a specific point in the solution space. Such solutions are first refined using a simple hill-climbing, accepting new solutions i.f.f. they dominate the archived ones. The hill-climbing refinement is performed $I$ times, then $HL$ non-dominated solutions are stored in the archive, using clustering to reduce its size, if needed. Then, Pareto-dominance relationship between newly generated solutions and the archived ones is evaluated, and whether a solution is accepted as part of the archive depends not only on the dominance relationship, but also on the current temperature of the "matter". Indeed, perturbations that increase the energy of a particular solution are still accepted at the beginning of the search procedure. Nevertheless, the probability of accepting such perturbations is gradually decreased as the temperature decreases; hence, all state transitions are bound to improve solutions.

As mentioned, the AMOSA heuristic adopts the *amount of domination* concept to compute the acceptance probability of new solutions. Given two solutions, say

$x$ and $y$, the amount of domination is defined in (3.8), where $M$ is the number of fitness-functions and $R_i$ is the range of the i-th fitness-function, as defined in (3.9). Please note that the latter may be not known a-priori.

$$\Delta_{dom_{x,y}} = \prod_{i=1,\, f_i(x) \neq f_i(y)}^{M} \frac{|f_i(x) - f_i(y)|}{R_i} \tag{3.8}$$

$$R_i = \max\{f_i(\cdot)\} - \min\{f_i(\cdot)\} \tag{3.9}$$

While the "matter" temperature is above $T_{min}$, in order to produce new candidate solutions $\eta$, the heuristic randomly $\rho$ from the archive, and alters its distinctive parameters. Then, the amount of domination of $\eta$ is checked against its parent and solutions from the archive. Three different scenarios may arise:

1. $\eta$ is dominated by its parent $\rho$, and also by $k$ other solutions from the archive; in this scenario, $\eta$ is not discarded, rather accepted as new $\rho$ with a probability given by Equation (3.10), where $T$ is the current temperature, and $\Delta_{dom_{avg}}$ is the *average amount of domination*.

$$\begin{aligned} P &= \frac{1}{1 + e^{T \times \Delta_{dom_{avg}}}} \\ \Delta_{dom_{avg}} &= \frac{\sum_{i=1}^{k} \Delta_{dom_{i,\eta}} + \Delta_{dom_{\rho,\eta}}}{k+1} \end{aligned} \tag{3.10}$$

2. $\eta$ and $\rho$ are non-dominating w.r.t. each other; in this case, based on the amount of domination with archived solutions, three scenarios may arise:

   (a) $\eta$ is dominated by $k, k \geq 1$ different archived solutions; in this scenario, $\eta$ is not discarded, rather accepted as new $\rho$ with a probability given by Equation (3.11);

$$\begin{aligned} P &= \frac{1}{1 + e^{T \times \Delta_{dom_{avg}}}} \\ \Delta_{dom_{avg}} &= \frac{\sum_{i=1}^{k} \Delta_{dom_{i,\eta}}}{k} \end{aligned} \tag{3.11}$$

(b) $\eta$ is non-dominating w.r.t archived solutions; $\eta$ is accepted as new $\rho$ and added to the archive;

(c) $\eta$ is dominating w.r.t $k, k \geq 1$ archived solutions; $\eta$ is accepted as new $\rho$ and added to the archive; in addition, dominated solutions are removed from the archive.

3. $\eta$ dominates $\rho$; in this case, based on the amount of domination with archived solutions, three scenarios may arise:

(a) $\eta$ is dominated by $k, k \geq 1$ different archived solutions; this situation may arise only if $\rho$ is not part of the archive; a different point, minimizing Equation (3.12), is selected from the archive as new $\rho$ with probability given by (3.13); otherwise, $\eta$ is accepted as new $\rho$;

$$\Delta_{dom_{min}} = \min_{i}\{\Delta_{dom_{i,\eta}}\} \tag{3.12}$$

$$P = \frac{1}{1 + e^{\Delta_{dom_{min}}}} \tag{3.13}$$

(b) $\eta$ is non-dominating w.r.t archived solutions; if $\rho$ belongs to the archive, it is discarded, then $\eta$ is accepted as new $\rho$ and added to the archive;

(c) $\eta$ is dominating w.r.t $k, k \geq 1$ archived solutions; $\eta$ is accepted as new $\rho$ and added to the archive; in addition, dominated solutions are removed from the archive.

The above process is repeated $I$ times for each temperature, then the latter is reduced according to the cooling factor $\alpha$ until the $T_{min}$ is attained.

#### 3.1.2.4 MOP modeling: identifying decision-variables and suitable fitness functions

Modeling a specific optimization problem is not trivial, and no general rules exist. Anyway, taking into account the technique used to generate the approximate variants definitely helps in at least identifying the decision variables of the problem. Indeed, the latter find natural correspondence in the configuration parameters introduced to govern the degree of approximation.

As already discussed, the identification of suitable decision-variables is only the first step to complete to define a MOP-based DSE. Indeed, we need to also define fitness-function driving the DSE. In particular, we must assess the error entailed by the approximations. Hence, we need to pinpoint an appropriate error metric in order to define a suitable error fitness-function to minimize. Unfortunately, when using the AxC paradigm, defining an appropriate error metric is of major concern, and it is usually not a trivial task. Therefore, the error-metric is usually selected case-by-case. Anyway, for some applications, the choice of error metric is obvious, if not outright forced, by the application-domain. The classification accuracy-loss, for instance, is a meaningful error metric for either DNNs and DT MCSs applications, while the PSNR or the SSIM are common error metrics in the image-processing field.

For what pertains to hardware-requirements, in order to accurately take into account the resource savings in the DSE, we should measure area, power-consumption and maximum operating frequency of the explored approximate configurations. Unfortunately, this would require the synthesis and simulation of each approximate configuration explored during the DSE, which is definitely a time-consuming process. Therefore, we propose a model-based hardware-resources estimation to drive the DSE. This has to take into account the impact of the selected approximation technique on the final hardware implementation, in order to provide a faithful estimation, albeit not accurate. Defining such a model is not straightforward. Albeit removing some parts of an arithmetic circuit, for instance, undoubtedly leads to specific gains in terms of area/energy, model-based hardware-requirement estimation becomes trickier when the approximation has to be tailored to the application, and savings evaluated in the application's context, since they depend on the specific implementation.

### 3.1.3 Summary

For the reader convenience, Figure 3.2 summarizes the proposed methodology.

Starting from the model of the application to be approximate, an automatic approximation engine is adopted to generate configurable approximate variants. The latter may either be generated from scratch starting from the model, or resulting from alterations of the model itself. Furthermore, for each approximate portions, approximate variants

allow to selectively adjust the degree of introduced approximation, through the use of convenient configuration parameters. The value for such parameters leading to optimal trade-offs between quality of results and performance gains is searched through a MOP-based DSE. The latter is solved using a suitable heuristic, e.g., either NSGA-II or AMOSA, while minimizing error entailed by the approximation and, at the same time, hardware-requirements. At the end of the DSE, resulting non-dominated approximate configurations are adopted to suitably shape a configurable accelerator implementation. A synthesis tool can, then, be used to implement the latter on a technology of choice.

In the next chapters we apply the proposed methodology to several applications from different domain, including generic logic, image-processing and machine-learning applications.

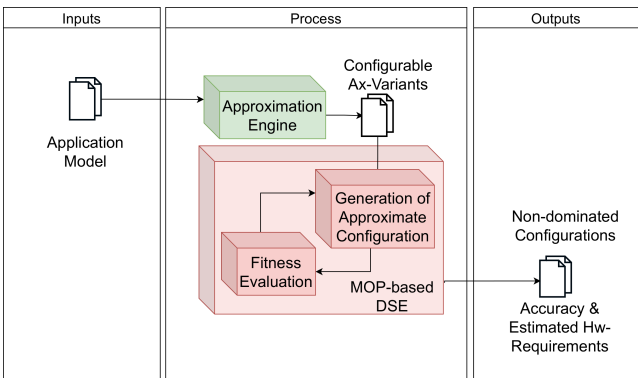



Figure 3.2: Workflow of the proposed methodology

# Chapter 4

# Combinational logic case-studies

In this Chapter, we apply our methodology to the design of combinational logic circuits, i.e., those that typically constitute building-blocks for larger, more complex, designs.

Concerning circuits approximation, a variety of techniques has been proposed in the scientific literature, including both frequency/voltage over-scaling and functional approximation [147, 154], though the latter are generally application-dependent, their usage requires designers to have precise insights on the application being approximate, and most of them tackle the DSE either combining multiple design objectives in a single-objective optimization problem, or optimizing for a single parameter while keeping constant the remaining ones. Hence, only a small fraction of the approximate configurations set is taken into consideration, often yielding results that can certainly be further improved.

Conversely, the methodology we presented in Chapter 3 is systematic, automatic and application-independent. In the following, we will discuss how to automatically generate approximate variants for combinational circuits, and how to select the best trade-offs between error and savings through the use of a MOP-based DSE. In particular, in order to generate approximate variants, we exploit the AIG representation of digital

circuits, and we resort to resort recent works concerning SAT and Exact Synthesis (ES) to build size-optimum approximate AIG cuts replacements. Then, MOP-based DSE selects k-feasible cuts to be replaced within the AIGs. In order to converge towards a Pareto-front, we adopt the AMOSA heuristic, with accuracy-loss and silicon-area minimization being the fitness-functions driving the DSE.

In the following, we first provide the reader with the needed background on AIG and ES in Section 4.1, then we discuss the generation of approximate variants in Section 4.2, and several aspects concerning MOP-based DSE in Section 4.3, including MOP-modeling and fitness-functions to drive the DSE. Finally, in Section 4.4 we present experimental results.

## 4.1 Preliminary technical background

In this section, we firstly provide the reader with the needed technical background, introducing the AIG formalism and discussing the ES problem.

### 4.1.1 And-Inverter Graphs

An AIG [122] is a direct a-cyclic graph in which there are PI nodes, which have no incoming edges, and logic-AND nodes, which have two incoming edges. The latter represent physical connections between nodes, and they can be marked as complemented or not.

Consider a Boolean function $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$ and its set of input variables $\{x_1, \ldots, x_n\}$. An AIG is formally defined as the set of nodes $\{x_{n+1}, \ldots, x_{n+r}\}$ combined accordingly to Equation (4.1), with $r$ being the AIG size i.e. its number of nodes, $s_{1i} < s_{2i} < i$ being indexes of the nodes and $p^{1i}, p^{2i}$ being the polarity of the incoming edges of the i-th node. Conventionally, the polarity of complemented edges is 0.

$$x_i = x_{s_{1i}}^{p_{1i}} \wedge x_{s_{2i}}^{p_{2i}} \tag{4.1}$$

An AIG is said to realize the Boolean function $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$ i.f.f. Equation (4.2) is

satisfied. Nevertheless, being the AIG representation non-canonical, the same Boolean function can be realized by multiple different AIGs.

$$f_i = x_{s_i}^{p_i} \qquad i \in [1, m]\,,\ s_i \in [n+1, n+r] \tag{4.2}$$

A set $\{x_{i_1}, \ldots, x_{i_k}\}$ of nodes is said to be a *path* of length $k$ if (a) $i_1 < n$, i.e., the set starts from a PI or from a constant, (b) $i_j \in$ *children*$(i_{j+1})$, i.e., the set is an ordered sequence of interconnected nodes, and (c) $\exists\ l \in [1, m] : s_l = i_k$, i.e., the sequence ends in a PO. The longest path is called the *critical path*. Consider the set of paths ending in a node $x_i$, described by Equation (4.3): the $(i, L)$ pair – consisting of the root node $x_i$ and the set of leaf nodes $L \subseteq \{x_1, \ldots, x_{n+s}\}$ – defines a *cut* i.f.f. (a) $\forall p \in paths(i),\ p \wedge L \neq \emptyset$ i.e., all paths to $x_i$ contain at least a leaf node from $L$, and (b) $\forall l \in L\ \exists p \in paths(i) : l \in p$ i.e. each leaf in $L$ is at least within a path to $x_i$. A cut is *k-feasible* if $|L| \leq k$, where $|L|$ is the size of $L$.

$$paths(i) = \bigcup_{j \in children(i),\ j \neq 0} \{paths(j),\ i\} \tag{4.3}$$

The set of all k-feasible cuts having $x_i$ as the root node is recursively defined by Equation (4.4).

$$cuts_k\,(i) = \begin{cases} \emptyset & i = 0 \\ i & i \in [1, n] \\ cuts_k\,(s_{1i}) \oplus_k cuts_k\,(s_{2i}) & i \in [n+1, n+r] \end{cases} \tag{4.4}$$

The $\oplus_k$ operator in (4.4) is the saturating union over all the combinations of subsets extracted from two sets, defined by (4.5).

$$M_1 \oplus_k M_2 = \{m_1 \cup m_2 : |m_1 \cup m_2| \leq k,\ m_1 \in M_1, m_2 \in M_2\} \tag{4.5}$$

### 4.1.2 The Exact Synthesis Problem

Essentially, the ES problem consists of finding a combinational circuit that realizes a given Boolean function specification, and that turns out optimal w.r.t. some cost criteria, which is usually the number of nodes and/or the circuit depth. So far, its computational

complexity is unknown, although the minimum circuit size problem, of which the ES is an instance, has been extensively studied and efficient algorithms for it are considered to be unlikely [131]. Nevertheless, solutions for the ES problem can be efficiently found by solving the decision problem in Equation (4.6), which asks whether there exists an AIG of a given size $r$ and a given maximum depth $d$ that realizes a certain Boolean function $f$. In the following sections, we refer to Equation (4.6) as *HasAIG(f, r, d)*: it either returns an AIG of size $r$ and a polarity $p$ for the output node or *unsat* whether such an AIG does not exist.

$$\exists\{x_{n+1},\ldots,x_{n+r}\}, p \in [0,1] \ : \left(x_{n+r}^{p} = f\right) \wedge \left(l_{n+r} \leq d\right) \quad (4.6)$$

If the AIG depth is not taken into account, then (4.6) gets simplified as follows.

$$\exists\{x_{n+1},\ldots,x_{n+r}\}, p \in [0,1] \ : \left(x_{n+r}^{p} = f\right) \quad (4.7)$$

Progresses made in scientific literature allow SAT problems to be formulated as Satisfiability-Modulo Theory (SMT) problem, and solved in reasonable time [59]. We adapt the SMT problem formulation from [161] to AIGs and, although we present a formulation suitable for single-output Boolean function, its generalization to multiple outputs Boolean functions is effortless.

Consider an $n$-inputs Boolean function $f : \mathbb{B}^m \rightarrow \mathbb{B}$. In order to formulate the SMT problem, it is needed (i) introducing $\{s_{1i}, s_{2i}\}$ indexes and $\{p_{1i}, p_{2i}\}$ Boolean variables, for $i \in [n+1, n+r]$; (ii) enforcing constraint in Equation (4.8), in order to both forbid cycles and to define an ordering between nodes; (iii) encoding the logic-AND behavior of each node, as stated in Equation (4.9); (iv) encoding PIs connection and enforcing values propagation through the AIG, using Equation (4.10), and, finally (v) encoding the equivalence constraint and the actual function semantic (4.11).

$$s_{1i} < s_{2i} < i \qquad i \in [n+1, n+r] \quad (4.8)$$

$$b_i^{(t)} = a_{1i}^{(t)} \wedge a_{2i}^{(t)} \quad (4.9)$$

$$s_{ci} = j \Rightarrow a_{ci}^{(t)} = b_j^{(t)} \oplus \overline{p_{ci}} \qquad c = \{1,2\} \quad (4.10)$$

$$b_{n+r}^{(t)} = \overline{p} \oplus f(t) \quad (4.11)$$

This formulation makes use of the explicit function representation – i.e., $f$ is represented in terms of truth table values, for each of the possible $2^n$ input assignments. Moreover, in order to encode the behavior of the Boolean function for each input assignment, each node $i \in [n+1, n+r]$ is replicated once for each of the input vectors $t \in [0, \ldots, 2^n - 1]$. The $a_{1i}^{(t)}$, $a_{2i}^{(t)}$ and $b_i^{(t)}$ variables represents, respectively, the value of input signals for the i-th node and its output while the circuit input is set to the input vector $t \in [0, \ldots, 2^n - 1]$. The $b_{n+r}^{(t)}$ node, which is the node having the largest index, is the root node of the AIG i.e., the output node.

In order to solve the ES problem, we adapt the algorithm proposed in [161] to AIGs. Consider the $(x, p) \leftarrow HasAig(f, r)$ procedure, that, given a Boolean function $f$, returns an AIG of size $r$ if the latter exists, with $x$ being the AIG structural description and $p$ the output node's polarity. In order to solve the ES problem, a naive approach would increase $r$ as long as no AIG having that size exists, as shown in Algorithm 1.

**Algorithm 1:** ES algorithm from [161]

```
Function ExactAIG(f):
    Input: function f
    Output: AIG x, polarity p
    r ← 0;
    while true do
        aig ← HasAig(f, r);
        if aig ≠ nil then
            return aig;
        else
            r ← r + 1;
```

## 4.2 Catalog-based AIG rewriting

Consider a certain Boolean function $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$. The Algorithm 1 we discussed in Section 4.1.2 can effectively be exploited to generate approximate variants for a Boolean function $f$, by searching for AIGs that realize a different function $f' \neq f$, which is implemented by a smaller AIG w.r.t. $f$, yet it is acceptable according to some error metric. This would enable to obtain size-optimum approximates circuits

which also differ from the exact one in a very controlled way. Unfortunately, this naive approach is quite resource-intensive, since computational time needed to solve the ES problem in Equation (4.6) is reasonable only when considering functions with a fairly small number of inputs. Hence, it is hardly applicable to actual circuits. Nevertheless, the approach can effectively be applied to k-feasible cuts, since the latter are, in essence, $k$-inputs single-output Boolean functions.

Briefly, the general idea behind our approach is to enumerate k-feasible cuts for a given AIG, and, then, supersede carefully selected cuts with similar ones, exhibiting better performances. Anyway, a complete enumeration of the set of k-feasible cuts in Equation (4.4) is not feasible whether $k \geq 6$ [124]. Therefore, we exploit partial cut-enumeration algorithms, which are effectively adopted for FPGA synthesis in order to compute the graph of interconnected k-cuts or, alternatively, interconnected k-LUTs [124].

Thus, for each unique k-cut $c$ within the AIG $C$, we generate a catalog of approximate cuts, each of which is an approximate Boolean function at a predetermined Hamming distance from the considered cut. Approximate variants' generation for $C$ take place by substituting a given cut – or, alternatively, a LUT instance – using one of its approximate variants, picked from the catalog, and rewriting back the corresponding AIG. The relationship between an AIG size, in terms of both depth and number of nodes, and its hardware requirements in terms of critical path and cells suggests that whether the approximate circuit consists of fewer nodes, then its hardware requirements will be lower than the original circuit [124]. The catalog-generation procedure is detailed in Section 4.2.1.

For the sake of clarity, let us consider the AIG in Figure 4.1, which represents a 4-inputs-4-outputs Boolean function: the 4-LUT technology mapping results in the implementation depicted in Figure 4.2, which is equivalent to the set of 4-feasible cuts of the AIG in Figure 4.1. We introduce approximation by superseding a cut, which is equivalent to a 4-inputs-1-output Boolean function, with a similar Boolean function, and rewriting back the AIG. Let us consider, for instance, the cut having $o[3]$ as the root node and PIs as leaf nodes or, alternatively, the $LUT12$ in Figure 4.2. It implements the $o[3] = a[0] \wedge a[1] \wedge b[0] \wedge b[1]$ Boolean function, and it can be superseded by making use of a constant zero, which is a function at Hamming distance 1 w.r.t the function implemented by $LUT12$. Then, by rewriting back the AIG, it will result in

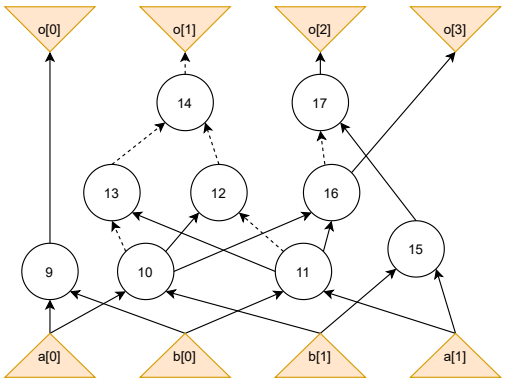



Figure 4.1: AIG of a 4-inputs-4-outputs Boolean function.

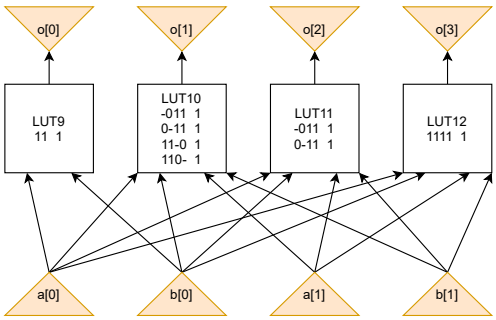



Figure 4.2: AIG of Figure 4.1, mapped to 4-LUT.

the Boolean function whose AIG is depicted in Figure 4.3. Now, depending on the final application, a suitable error metric has to be selected, and error assessment performed.

### 4.2.1 Catalog generation

The catalog-generation procedure is detailed in Algorithm 2. Starting from f_of(node), i.e., the exact specification of each LUT in the considered circuit, we progressively increase the Hamming distance between the function being implemented by original cut and the approximate one, while performing ES. The procedure stops when, due to the approximation itself, the synthesis becomes trivial, i.e., it results in a catalog entry of size zero, meaning the approximate Boolean function requires no AIG nodes to be

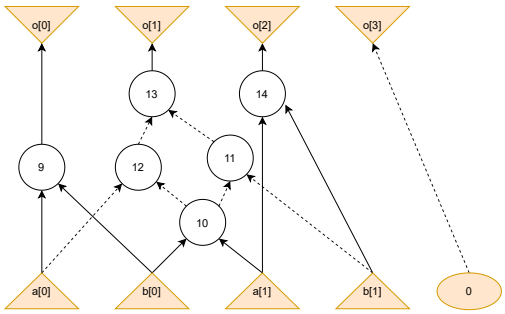



Figure 4.3: Approximate configuration of the AIG of Figure 4.1

implemented.

**Algorithm 2:** Catalog generation algorithm

```
Function CreateCatalog(C_lut):
    Input: C_lut: LUT-mapped circuit
    Output: catalog: LUT catalog
    catalog ← ∅;
    foreach node ∈ C_lut do
        if f_of(node) ∉ catalog then
            catalog ← catalog ∪ node;
            d ← 0;
            do
                catalog[f_of(node)][d] ← ExactAIG(f_of(node), d);
                d ← d + 1;
            while gates(catalog[f_of(node)][d]) > 0;
    return catalog;
```

The catalog-generation procedure requires the $HasAig(f, r)$ to be slightly modified, in order to accommodate for the degree of approximation – i.e., the Hamming distance – $d$ to be introduced. In particular, let us consider a $f : \mathbb{B}^n \rightarrow \mathbb{B}$, we implement the function semantic defined in Equation (4.12), which means that at most $d$ distinct elements belonging to the input set do satisfy the inequality or, in other words, that at least $2^n - d$ input-output correspondences of the exact behavior are preserved.

$$\exists_{\leq d} t \in \{0, \ldots, 2^n - 1\} \ : \ b_{n+r}^{(t)} \neq \overline{p} \oplus f(t) \tag{4.12}$$

Algorithm 3 reports a pseudocode implementation of such a procedure.

**Algorithm 3:** ES algorithm for approximate AIGs

```
Function ExactAIG(f, d):
    Input: function f
    Output: AIG x, polarity p
    r ← 0;
    while true do
        aig ← HasAig(f, r, d);
        if aig ≠ nil then
            return aig;
        else
            r ← r + 1;
```

### 4.2.2 Scalability issues

Since the catalog-generation procedure does depend neither on the specific circuit being approximated nor on the output quality constraints, it is possible to enumerate all the Boolean k-inputs functions – i.e., all the possible configurations of k-LUTs – in order to pre-compute the catalog. This would dramatically improve the scalability of our approach, since it allow reusing the catalog several times, avoiding having to repetitively solve the same ES problem while performing approximation of different Boolean functions.

However, such a generic catalog-generation procedure may be enormously time-consuming, depending on the chosen $k$, since the catalog size, i.e., the number of k-inputs-1-output Boolean functions, grows rapidly with $k$. Indeed, for each of the $2^{2^k}$ possible catalog entries, the set of all functions at $d \in [1, 2^k]$ Hamming distance must be recorded. Thus, the full size of the catalog is $2^{2^k} \times \sum_{i=0}^{2^k} \binom{2^k}{i}$.

Anyway, a pragmatic approach to outflank such an issue would consist of incrementally build the catalog, while considering different circuits.

Concerning the SMT problem formulation, properties of the ES problem can be exploited to shorten the computational time. Indeed, the *HasAig(f, r)* ES problem has a monotonic behavior, i.e., if *HasAig(f, r)* can be satisfied for a given $r$, then *HasAig(f, r+1)* can also be satisfied [161]. Thus, linear search can ideally be superseded using binary search, since the upper bound for the number of gates required by the synthesis of an arbitrary Boolean function always exists [118]. Though, proving that the

*HasAig(f, r)* cannot be satisfied is much more time-consuming than proving *HasAig(f, r+1)* can be satisfied [161], hence binary search does not necessarily perform better than linear one. In addition, this observation paves the way for a heuristic approach to circumvent time-consuming *unsatisfiability* trials: if *HasAIG(f, r)* cannot be solved within a given time budget, the heuristic behaves as if such a problem is *unsatisfiable*.

## 4.3 Design-space exploration

For the reader's convenience, we summarize the variant generation procedure. Since circuit specifications are typically encoded in Hardware Description Language (HDL) rather than AIG, a preliminary analysis is carried out in order to obtain an AIG representation of the given circuit, according to the implemented Boolean function. Then, k-LUT mapping is performed, in order to enumerate k-feasible cuts. Thus, for each unique k-LUT, a catalog of variants is generated as discussed in Section 4.2: constraints (4.8), (4.9) and (4.10) are encoded to solve the approximate ES problem (4.12) while increasing Hamming distance $d$ until, due to the approximation itself, the synthesis becomes trivial.

Once the catalog has been built, the main challenge is to find the combination of cut-replacements leading to Pareto-optimal trade-offs between error and savings, i.e., to perform a DSE. In this case-study, we adopt the AMOSA [19] searching algorithm, that is deeply discussed in Section 3.1.2.3. The AMOSA heuristic orchestrates the DSE: k-LUT nodes constitute the set of decision variables of the MOP, their indexes are assigned according to the topological ordering defined by the underlying graph, and their domain is given by catalog entries. Starting from a randomly chosen archived solution, the AMOSA selects a random LUTs and replace it using a suitable element taken from the catalog, then fitness-functions are computed to state the Pareto-dominance relationship between the altered configuration and archived solutions. As discussed in Section 4.2.2, catalog entries are organized and stored in a database so that they can be subsequently reused. Moreover, properties of the ES problem are exploited in order to improve scalability.

As we mentioned, fitness-functions driving the DSE are error and silicon-area minimization. As far as silicon-area is concerned, we resort to a model-based gain

estimation to drive the DSE. In particular, we estimate the silicon-area requirements from the number of AIG nodes, since the relationship between the latter and hardware requirements – in terms of critical path and LUTs or standard cells – has been empirically proven in [124]. We evaluate both the number of nodes and the depth for a given approximate configuration on Functionally-Reduced And-Inverter Graphs (FRAIGs) [123].

For what pertains to error-metrics, the one to be used strictly depends on what kind of circuits is the final target, since a suitable error metric must be selected. In the following, we discuss experimental results while targeting generic combinational logic circuits and arithmetic circuits. In the former case, we adopt the error probability metric, which is defined by Equation (2.5), while in the latter case we resort to the AWCE metric, as defined by Equation (2.1). Concerning the error-assessment method, we resort to simulation-based assessment. Anyway, using exhaustive test patterns simulation is often prohibitively time-consuming, therefore, we configured our tool to perform exhaustive simulations only whether the amount of PIs is less or equal to than 13, since it would involve less than 10000 test vectors. Conversely, for circuits having more inputs, we resort to an error-sampling method, applying a sequence of pseudo-random input vectors to each targeted approximate variant. Concerning the error-probability, in order to estimate the actual error, we resort to Equation (4.13), which provides a 99.7% confidence-level for the actual error given the number of test patterns $N_s$ and the measured error $R_S$ [87]. During DSE, we do set $N_s = 10000$. No error threshold has been set during this experimental campaign, even though our approach allows us to impose constraints on both maximum error and gate-count.

$$R_E = R_S + \frac{4.5}{N_S} \times \left(1 \pm \sqrt{1 + \frac{4}{9} \times N_S \times R_S \times (1 - R_S)}\right) \quad (4.13)$$

## 4.4 Experimental results

The approach presented in Section 4.2 is implemented as a plug-in of the Yosys synthesis tool [178], using the Boolector SMT solver [133] to solve the ES problem during the catalog-generation procedure. The latter is performed targeting 4-feasible and 6-feasible cuts, since 4-LUT and 6-LUT are currently the most frequent solutions

adopted in FPGA commercial fabric.

In order to evaluate the proposed methodology, we first considered a subset of the LGSynt91 benchmark [181], including both logic and arithmetic circuits. We also considered arithmetic circuits from [2] for further evaluation.

### 4.4.1 The LGSynt91 benchmark

As mentioned, the LGSynth91 benchmark is quite diversified, and comprises both generic combinational logic and arithmetic circuits. For this batch of experiments, we selected the error-probability as error metric and the number of AIG gates to estimate silicon-area requirements, as mentioned above. At the end of the DSE, the AMOSA heuristic provided several approximate configurations for each of the considered benchmark circuits. Figure 4.4 reports the Pareto-front estimation resulting from DSE for some of the circuits belonging to the considered benchmark.

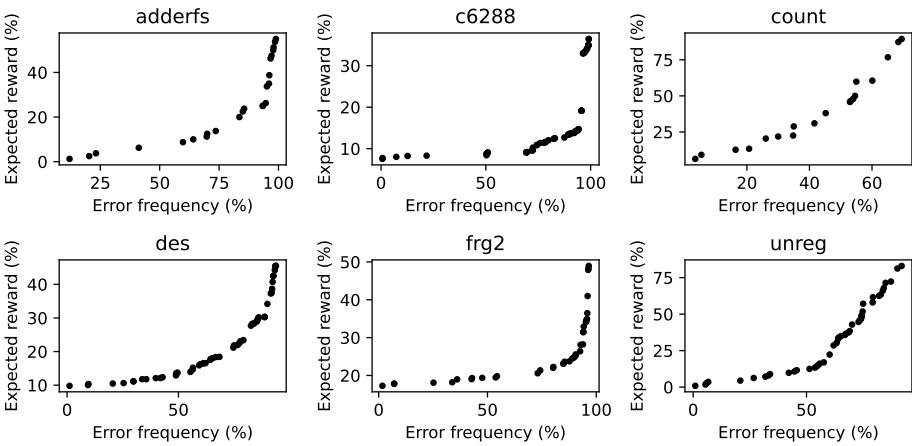



Figure 4.4: Pareto-fronts resulting from DSE for some of the circuits from the LGSynt91 benchmark.

In order to state the actual savings, we also performed FPGA synthesis and power analysis targeting a mid-range Xilinx Artix 7 FPGA. Experimental results are reported in Table 4.1. Such a table also report the rough computational-time needed to perform the whole workflow on each of the considered circuits. Experiments have been con-

ducted on a host PC equipped with 16 GB of RAM and an Intel i7-3770 CPU running at 3.9 GHz.

For each of the considered benchmark circuits we report: the circuit name and its type (i.e., whether the circuit is pure logic or arithmetic), the gate count, the LUT count, the estimated power consumption and related savings for either the exact (i.e., non-approximated) circuit, the minimum-error and the minimum area approximate configurations.

Although circuits in the considered benchmark do not exhibit the same error resiliency, the proposed methodology is able to exploit relaxed constraints on the quality of the output, allowing for significant area savings, both for smaller and larger circuits. In addition, larger circuits in terms of AIG nodes, where the set of k-feasible cuts is quite large – hence, a larger number of variants and approximate configurations can be identified – the optimization heuristic has much more room to operate, yielding substantial savings. The same cannot be said for circuits of modest or small size, where the results are still substantial, but much less eye-catching.

As it is easy to grasp, a reduction in power consumption is expected a consequence of area savings; therefore, we also performed power analysis while resorting to workload-based simulations for all the Pareto-optimal solutions of the considered benchmark circuits. Results, still reported in Table 4.1, show that a significant amount of savings is achieved for almost all the considered circuits, depending on their error resiliency and the corresponding approximation degree. Power savings are attributable to two different contributions: on the one hand, the reduced area of approximate circuits directly correlate to a reduced static power consumption, while, on the other hand, the lower switching activity approximate cuts exhibit w.r.t. exact ones – which also reflects on LUTs implementing the approximate circuits – lead to a reduction of the dynamic term.

### 4.4.2 Arithmetic circuits

Being the metric we selected while experimenting on the LGSynt91 benchmark suitable for generic logic circuits rather than arithmetic ones, the achieved savings for those

Table 4.1: Experimental result on the LGSynt91 Benchmark

| | | | Exact Solution | | | Min. Error Approx. Configuration | | | | | | Min. Area Approx. Configuration | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Circuit** | **Type** | **Comp. Time** | **Gates** | **FPGA LUTs** | **Power Consumption (mW)** | **Error** | **Expected Reward (%)** | **FPGA LUTs** | **Area Savings (%)** | **Power Consumption (mW)** | **Power Savings (%)** | **Error** | **Expected Reward (%)** | **FPGA LUTs** | **Area Savings (%)** | **Power Consumption (mW)** | **Power Savings (%)** |
| 9symml | Count Ones | 5s | 43 | 31 | 0.067 | 1.95E-03 | 58.16 | 3 | 90.32 | 0.061 | 8.96 | 1.56E-02 | 80.75 | 2 | 93.55 | 0.059 | 11.79 |
| alu2 | ALU | 16m | 335 | 146 | 0.092 | 8.00E-03 | 2.29 | 112 | 23.29 | 0.091 | 1.09 | 5.51E-01 | 49.03 | 17 | 88.36 | 0.082 | 10.87 |
| alu4 | ALU | 7m | 681 | 234 | 0.107 | 7.89E-04 | 3.82 | 230 | 1.71 | 0.107 | 0.00 | 8.00E-01 | 34.86 | 24 | 89.74 | 0.080 | 25.23 |
| apex6 | Logic | 6m | 452 | 185 | 0.340 | 5.00E-03 | 9.12 | 169 | 8.65 | 0.326 | 4.12 | 8.50E-01 | 24.96 | 136 | 26.49 | 0.301 | 11.47 |
| b1 | Logic | 1s | 13 | 3 | 0.073 | 1.25E-01 | 37.50 | 3 | 0.00 | 0.071 | 2.74 | 2.50E-01 | 62.50 | 2 | 33.33 | 0.069 | 5.48 |
| b9 | Logic | 10s | 125 | 36 | 0.111 | 5.30E-02 | 55.05 | 19 | 47.22 | 0.093 | 16.22 | 3.83E-01 | 87.16 | 6 | 83.33 | 0.076 | 31.53 |
| C17 | Logic | 1s | 6 | 2 | 0.067 | 6.25E-02 | 28.57 | 2 | 0.00 | 0.067 | 0.00 | 5.89E-01 | 77.95 | 1 | 50.00 | 0.065 | 2.99 |
| C432 | Decoder | 1m | 160 | 102 | 0.108 | 1.40E-02 | 41.42 | 22 | 78.43 | 0.079 | 26.85 | 4.78E-01 | 69.87 | 7 | 93.14 | 0.067 | 37.96 |
| C6288 | Multiplier | 9m | 2406 | 754 | 0.472 | 6.48E-03 | 7.28 | 731 | 3.05 | 0.471 | 0.21 | 9.82E-01 | 34.07 | 522 | 30.77 | 0.298 | 36.86 |
| c8 | Logic | 12s | 164 | 26 | 0.127 | 1.64E-02 | 62.99 | 12 | 53.85 | 0.101 | 20.47 | #REF! | 77.95 | 11 | 57.69 | 0.095 | 25.20 |
| cc | Logic | 9s | 47 | 16 | 0.107 | 5.75E-03 | 24.24 | 14 | 12.50 | 0.105 | 1.87 | 5.92E-01 | 74.24 | 6 | 62.50 | 0.099 | 7.48 |
| cm138a | Logic | 12s | 17 | 8 | 0.090 | 1.56E-02 | 23.08 | 6 | 25.00 | 0.090 | 0.00 | 3.59E-01 | 76.92 | 3 | 62.50 | 0.063 | 30.00 |
| cm150a | Logic | 7s | 69 | 9 | 0.066 | 1.48E-03 | 4.35 | 8 | 11.11 | 0.066 | 0.00 | 2.36E-01 | 97.83 | 1 | 88.89 | 0.062 | 6.06 |
| cm151a | Logic | 8s | 33 | 4 | 0.068 | 4.00E-02 | 3.85 | 4 | 0.00 | 0.068 | 0.00 | 3.45E-01 | 88.46 | 3 | 25.00 | 0.066 | 2.94 |
| cm152a | Logic | 6s | 30 | 3 | 0.066 | 1.60E-02 | 58.62 | 0 | 100.00 | 0.059 | 10.61 | 1.60E-02 | 68.97 | 1 | 66.67 | 0.060 | 9.09 |
| cm162a | Logic | 1m | 43 | 10 | 0.075 | 2.17E-03 | 23.81 | 9 | 10.00 | 0.075 | 0.00 | 1.60E-01 | 90.48 | 3 | 70.00 | 0.066 | 12.00 |
| cm163a | Logic | 5m | 42 | 8 | 0.074 | 1.08E-04 | 19.05 | 7 | 12.50 | 0.073 | 1.35 | 3.41E-01 | 90.48 | 4 | 50.00 | 0.070 | 5.41 |
| cm42a | Logic | 3s | 17 | 10 | 0.071 | 6.25E-02 | 30.00 | 7 | 30.00 | 0.067 | 5.63 | 3.13E-01 | 70.00 | 3 | 70.00 | 0.064 | 9.86 |
| cm82a | Logic | 3s | 27 | 3 | 0.076 | 1.25E-01 | 10.00 | 3 | 0.00 | 0.076 | 0.00 | 6.88E-01 | 80.00 | 2 | 33.33 | 0.069 | 9.21 |
| cm85a | Logic | 2s | 38 | 8 | 0.069 | 3.00E-02 | 45.45 | 5 | 37.50 | 0.067 | 2.90 | 3.46E-01 | 93.18 | 1 | 87.50 | 0.065 | 5.80 |
| cmb | Logic | 3s | 41 | 9 | 0.063 | 7.89E-04 | 15.09 | 9 | 0.00 | 0.063 | 0.00 | 5.99E-01 | 96.23 | 1 | 88.89 | 0.063 | 0.00 |
| count | Counter | 4min | 143 | 30 | 0.108 | 3.50E-02 | 5.63 | 29 | 3.33 | 0.108 | 0.00 | 6.84E-01 | 87.32 | 10 | 66.67 | 0.098 | 9.26 |
| cu | Logic | 3s | 48 | 15 | 0.074 | 1.60E-02 | 9.62 | 13 | 13.33 | 0.071 | 4.05 | 4.97E-01 | 90.38 | 3 | 80.00 | 0.062 | 16.22 |
| decod | Decoder | 12s | 22 | 16 | 0.070 | 3.13E-02 | 12.00 | 14 | 12.50 | 0.069 | 1.43 | 5.94E-01 | 88.00 | 1 | 93.75 | 0.061 | 12.86 |
| des | Encription | 7.5h | 4000 | 1020 | 1.104 | 1.10E-02 | 9.53 | 519 | 49.12 | 0.888 | 19.57 | 9.38E-01 | 45.49 | 244 | 76.08 | 0.688 | 37.68 |
| example2 | Logic | 10s | 277 | 88 | 0.169 | 6.48E-03 | 14.74 | 79 | 10.23 | 0.163 | 3.55 | 9.50E-01 | 48.72 | 25 | 71.59 | 0.126 | 25.44 |
| f51m | Arithmetic | 5s | 43 | 12 | 0.099 | 3.13E-02 | 64.52 | 4 | 66.67 | 0.077 | 22.22 | 3.75E-01 | 78.23 | 4 | 66.67 | 0.072 | 27.27 |
| frg1 | Logic | 16s | 105 | 120 | 0.075 | 2.00E-03 | 64.18 | 2 | 98.33 | 0.065 | 13.33 | 1.63E-01 | 69.27 | 2 | 98.33 | 0.063 | 16.00 |
| frg2 | Logic | 23m | 1004 | 225 | 0.354 | 1.70E-02 | 16.43 | 214 | 4.89 | 0.335 | 5.37 | 9.65E-01 | 48.95 | 69 | 69.33 | 0.203 | 42.66 |
| i1 | Logic | 3m | 46 | 16 | 0.096 | 3.25E-02 | 28.21 | 14 | 12.50 | 0.094 | 2.08 | 5.52E-01 | 84.62 | 6 | 62.50 | 0.082 | 14.58 |
| i3 | Logic | 10s | 90 | 70 | 0.093 | 3.75E-02 | 42.06 | 2 | 97.14 | 0.065 | 30.11 | 5.72E-01 | 84.92 | 3 | 95.71 | 0.062 | 33.33 |
| i5 | Logic | 3m | 285 | 75 | 0.270 | 7.20E-02 | 3.49 | 74 | 1.33 | 0.270 | 0.00 | 7.56E-01 | 26.03 | 68 | 9.33 | 0.250 | 7.41 |
| i6 | Logic | 11s | 340 | 67 | 0.902 | 8.70E-03 | 13.31 | 67 | 0.00 | 0.342 | 62.08 | 3.52E-01 | 40.25 | 52 | 22.39 | 0.289 | 67.96 |
| i7 | Logic | 5m | 471 | 67 | 0.367 | 2.84E-02 | 27.17 | 67 | 0.00 | 0.299 | 18.53 | 6.79E-01 | 55.51 | 43 | 35.82 | 0.255 | 30.52 |
| i8 | Logic | 17m | 1831 | 299 | 0.443 | 3.58E-03 | 17.42 | 209 | 30.10 | 0.349 | 21.22 | 2.56E-01 | 50.43 | 50 | 83.28 | 0.153 | 65.46 |
| i9 | Logic | 9m | 522 | 208 | 0.423 | 8.00E-03 | 46.10 | 77 | 62.98 | 0.255 | 39.72 | 1.28E-01 | 58.73 | 19 | 90.87 | 0.099 | 76.60 |
| pcler8 | Logic | 1min | 84 | 29 | 0.105 | 3.00E-02 | 34.34 | 18 | 37.93 | 0.103 | 1.90 | 5.64E-01 | 70.71 | 3 | 89.66 | 0.061 | 41.90 |
| pm1 | Logic | 22s | 39 | 16 | 0.083 | 1.40E-02 | 16.98 | 16 | 0.00 | 0.083 | 0.00 | 5.35E-01 | 92.45 | 9 | 43.75 | 0.075 | 9.64 |
| rot | Logic | 8m | 691 | 204 | 0.379 | 3.00E-03 | 24.93 | 136 | 33.33 | 0.336 | 11.35 | 4.05E-01 | 32.10 | 112 | 45.10 | 0.330 | 12.93 |
| sct | Logic | 1m | 91 | 19 | 0.099 | 2.51E-02 | 45.39 | 15 | 21.05 | 0.098 | 1.01 | 4.57E-01 | 76.60 | 10 | 47.37 | 0.088 | 11.11 |
| term1 | Logic | 52m | 358 | 52 | 0.082 | 1.87E-02 | 34.15 | 45 | 13.46 | 0.080 | 2.44 | 6.66E-01 | 59.41 | 8 | 84.62 | 0.073 | 10.98 |
| ttt2 | Logic | 5m | 200 | 37 | 0.122 | 2.01E-01 | 48.95 | 23 | 37.84 | 0.104 | 14.75 | 3.57E-01 | 56.76 | 21 | 43.24 | 0.101 | 17.21 |
| unreg | Logic | 2m | 97 | 16 | 0.119 | 9.45E-03 | 0.89 | 16 | 0.00 | 0.119 | 0.00 | 8.72E-01 | 72.32 | 14 | 12.50 | 0.094 | 21.01 |
| x2 | Logic | 7s | 42 | 12 | 0.077 | 6.20E-02 | 50.88 | 6 | 50.00 | 0.073 | 5.19 | 4.94E-01 | 82.46 | 5 | 58.33 | 0.068 | 11.69 |
| x3 | Logic | 7m | 715 | 178 | 0.339 | 1.17E-02 | 15.12 | 137 | 23.03 | 0.283 | 16.52 | 5.50E-01 | 24.57 | 122 | 31.46 | 0.306 | 9.73 |
| x4 | Logic | 4m | 369 | 86 | 0.250 | 3.84E-02 | 28.16 | 61 | 29.07 | 0.177 | 29.20 | 6.51E-01 | 43.37 | 57 | 33.72 | 0.160 | 36.00 |
| z4ml | Adder | 10s | 20 | 6 | 0.082 | 1.56E-02 | 71.29 | 1 | 83.33 | 0.066 | 19.51 | 4.69E-01 | 87.13 | 1 | 83.33 | 0.063 | 23.17 |

circuits – such as C6288, f51m and z4ml – are far modest when compared against pure-logical circuits. For this reason, we performed a second batch of experiments, using the AWCE defined by Equation (2.1) as metric for error assessment, involving the above-mentioned circuits and arithmetic circuits from [2]. For the error computation, each of the output bits was assigned a weight equal to the significance of the bit, i.e., the least significant bit has been assigned a weight of $2^0 = 1$, the next one a weight of $2^1 = 2$, and so on. As done for the LGSynt91, no constraints have been set. The reward fitness-function we adopted is still the number of AIG nodes.

First and foremost, for each of the circuit being considered, we report exact circuits requirements in terms of AIG gates, LUTs and power consumption, as provided by

Table 4.2: Experimental results while using the AWCE metric (2.1) on arithmetic circuits.

| Circuit | Comp. Time | Exact Circuit | | | Minimum Error Configuration | | | | | | Minimum Area Configuration | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Gates | FPGA LUTs | Power (mW) | Error | Expected Reward (%) | FPGA LUTs | Area Savings (%) | Power (mW) | Power Savings (%) | Error | Expected Reward (%) | FPGA LUTs | Area Savings (%) | Power (mW) | Power Savings (%) |
| C6288 | 17h | 2406 | 754 | 0.47 | 0 | 1.00 | 750 | 0.53 | 0.467 | 1.06 | 4.26E+09 | 37.00 | 484 | 35.81 | 0.28 | 41.31 |
| f51m | 3m | 43 | 12 | 0.10 | 1 | 30.00 | 8 | 33.33 | 0.09 | 5.05 | 7.50E+01 | 65.00 | 5 | 58.33 | 0.08 | 24.24 |
| z4ml | 2m | 20 | 6 | 0.08 | 1 | 8.00 | 6 | 0.00 | 0.08 | 0.00 | 7.00E+00 | 42.00 | 1 | 83.33 | 0.06 | 23.17 |
| 8x8 bits Dadda multiplier | 36m | 383 | 141 | 0.21 | 0 | 19.00 | 114 | 19.15 | 0.17 | 17.39 | 2.38E+02 | 43.00 | 39 | 72.34 | 0.10 | 51.69 |
| 8x8 bits array multiplier | 21m | 420 | 163 | 0.20 | 1 | 16.00 | 155 | 4.91 | 0.19 | 3.57 | 2.52E+02 | 45.00 | 48 | 70.55 | 0.10 | 46.94 |
| 8x8 bits Wallace multiplier | 40m | 398 | 149 | 0.20 | 0 | 12.00 | 132 | 11.41 | 0.17 | 13.93 | 2.50E+02 | 33.00 | 69 | 53.69 | 0.14 | 32.34 |
| 16 bits carry select adder | 3.5h | 138 | 44 | 0.18 | 0 | 2.00 | 44 | 0.00 | 0.17 | 5.68 | 4.10E+04 | 19.00 | 42 | 4.55 | 0.16 | 6.82 |
| 16 bits carry-skip adder | 3h | 128 | 39 | 0.17 | 0 | 4.00 | 39 | 0.00 | 0.17 | 1.16 | 1.78E+04 | 22.00 | 19 | 51.28 | 0.14 | 20.93 |
| 16 bits Han-Carlson adder | 4h | 120 | 38 | 0.17 | 1 | 1.00 | 38 | 0.00 | 0.16 | 5.23 | 6.17E+04 | 36.00 | 23 | 39.47 | 0.13 | 22.67 |
| 8 bits carry-lookahead adder | 6m | 54 | 24 | 0.61 | 1 | 9.00 | 16 | 33.33 | 0.11 | 81.64 | 3.43E+02 | 60.00 | 4 | 83.33 | 0.08 | 87.21 |
| 8 bits ripple carry adder | 3m | 59 | 19 | 0.12 | 1 | 4.00 | 13 | 31.58 | 0.12 | 0.00 | 1.45E+02 | 30.00 | 13 | 31.58 | 0.11 | 6.96 |
| 8 bits Han-Carlson adder | 4m | 53 | 18 | 0.12 | 1 | 3.00 | 13 | 27.78 | 0.12 | 0.00 | 3.77E+02 | 61.00 | 6 | 66.67 | 0.03 | 26.09 |

the Berkeley-ABC tool [29] using the *mcnc.genlib* library, and while targeting a mid-range Xilinx Artix 7 FPGA, respectively. For the sake of briefness, from the whole Pareto-front resulting from the DSE we only report solutions minimizing either one of the fitness-functions, i.e., the minimum-error and the minimum area approximate configurations. Furthermore, as done with the LGSynth91 benchmark, we also report the rough computational-time needed to perform the whole workflow on each fo the considered circuits.

As foreseeable, the use of a well-suited error metric is undoubtedly beneficial while considering arithmetic components for approximation. Indeed, concerning circuits from the LGSynth91 benchmark, using the AWCE rather than the error probability allows achieving significant savings while keeping the error bounded.

Concerning circuits from benchmark [2], our AIG-rewriting technique allows to carefully introduce approximation, while the MOP-based DSE provides a large variety of trade-offs between error and savings. Indeed, our methodology allows achieving minimum error configurations exhibiting almost no error yet providing savings ranging between 30-60%, while minimum area configurations provide higher savings, at the expense, obviously, of higher error.

Table 4.3: Comparison with results from [42] while using the error probability.

| Circuit | Results from [18] | | | | | Similar Error | | | | Similar gains | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Error freq. (%) | Gates | Delay | Area | Gates Ratio | Error freq. (%) | Gates | Delay | Area | Error freq. (%) | Gates | Delay | Area |
| alu2 | 45 | 236 | 32.2 | 570 | 0.91 | 55.1 | 203 | 8.9 | 137 | 10.5 | 332 | 19.4 | 765 |
| | 24 | 231 | 32.2 | 566 | 0.89 | 25.3 | 273 | 17.5 | 545 | | N/A | | |
| | 81 | 8 | 32.2 | 16 | 0.03 | 82.3 | 87 | 7,6 | 163 | | N/A | | |
| alu4 | 22 | 489 | 40 | 1172 | 0.94 | 24.5 | 527 | 20 | 1106 | 9.2 | 640 | 21.6 | 1304 |
| | 22 | 489 | 40 | 1172 | 0.94 | 24.5 | 527 | 20 | 1106 | 9.2 | 640 | 21.6 | 1304 |
| | 95 | 239 | 34 | 557 | 0.46 | 95.6 | 24 | 5.7 | 41 | 95.6 | 24 | 5.7 | 41 |
| cm163a | 43 | 15 | 4.1 | 25 | 0.44 | | N/A | | | 3.6 | 29 | 5.7 | 60 |
| | 21 | 18 | 3 | 36 | 0.53 | 23.6 | 15 | 3 | 28 | | N/A | | |
| | 88 | 20 | 4.7 | 41 | 0.59 | | N/A | | | 1.7 | 31 | 5.7 | 71 |
| count | 43 | 104 | 14.6 | 220 | 0.87 | 41.5 | 66 | 5.5 | 137 | 16 | 64 | 6.4 | 148 |
| | 24 | 53 | 3 | 110 | 0.44 | 26 | 48 | 5.1 | 118 | 45 | 43 | 4.6 | 98 |
| | 97 | 101 | 14.4 | 90 | 0.84 | | N/A | | | 16 | 64 | 6.4 | 148 |
| frg1 | 44 | 128 | 27.1 | 317 | 0.99 | | N/A | | | | N/A | | |
| | 16 | 126 | 27.1 | 313 | 0.98 | 10.5 | 0 | 0.9 | 1 | | N/A | | |
| | 56 | 128 | 27.1 | 317 | 0.99 | | N/A | | | | N/A | | |
| term1 | 38 | 113 | 11.1 | 249 | 0.80 | | N/A | | | | N/A | | |
| | 25 | 84 | 8.1 | 179 | 0.59 | 25.3 | 26 | 5.8 | 53 | 9.9 | 97 | 6.9 | 210 |
| | 99 | 73 | 11.1 | 150 | 0.51 | | N/A | | | 25.3 | 26 | 5.8 | 53 |
| unreg | 38 | 80 | 3.4 | 214 | 0.96 | 33 | 85 | 4.6 | 182 | 0.06 | 88 | 4.6 | 189 |
| | 0 | 83 | 3.4 | 227 | 1.00 | 0.9 | 111 | 5 | 224 | 0.9 | 111 | 5 | 224 |
| | 99 | 46 | 3.4 | 90 | 0.55 | 91 | 25 | 3.9 | 33 | 72 | 58 | 4.3 | 119 |
| x2 | 37 | 23 | 5.7 | 53 | 0.77 | 38 | 13 | 4 | 20 | | N/A | | |
| | 12 | 27 | 5.7 | 66 | 0.90 | 11 | 13 | 4 | 23 | | N/A | | |
| | 1 | 17 | 5.6 | 41 | 0.57 | 0 | 15 | 4 | 30 | | N/A | | |
| z4ml | 50 | 20 | 8.7 | 52 | 0.65 | 46 | 6 | 2.5 | 5 | | N/A | | |
| | 0 | 31 | 12.1 | 84 | 1.00 | 1.5 | 6 | 3.8 | 5 | | N/A | | |
| | 82 | 5 | 3.8 | 13 | 0.16 | | N/A | | | 46 | 5 | 3.8 | 3.8 |

### 4.4.3 Comparison with previous works

A comparable approach that exploits AIG-rewriting has been proposed by authors of [42]. Nevertheless, there are several differences between the two approaches: authors of [42] trivially replaces k-feasible cuts by making use of a stack-at fault with constant zero value while performing single-objective optimization targeting circuit area minimization with an error threshold. On the other hand, our approach is MOP-based, i.e., it simultaneously pursues both error and circuit area minimization, providing the designer with a comprehensive set of approximate configurations covering diversified trade-offs between error and gains.

For evaluation purpose, also authors of [42] considered a subset of the LGSynt91

benchmark, reporting approximate synthesis results provided by the Berkeley-ABC tool [] while using the *mcnc.genlib* library. Conversely, we consider FPGA synthesis. Therefore, in order to perform a fair comparison between results, we had to appropriately select, from the Pareto-front, configurations exhibiting error and gains similar to results from [42]. In addition, we also had to perform synthesis using the Berkeley-ABC tool [29] and the *mcnc.genlib* library. Table 4.3 reports results comparison while using the error probability as metric. Besides the error frequency, the gate count, delay and area requirements of approximate configurations resulting from [42], we also report a *gate ratio* column, i.e., the ratio between the gate count of an approximate configuration over the same for its non-approximate counterpart. For comparison purpose, we report approximate configurations obtained through our approach which present similar error frequency or similar savings w.r.t. approximate configurations from [42]. As the reader can observe, set the error degree, our method allows achieving better savings. On the other hand, set the savings, our method allows achieving approximate configurations exhibiting better quality of results.

Though, our approach allows introducing approximation in a less abrupt way, which leads to a more gradual degradation of the output quality. Moreover, the MOP-based approach allows effectively perform the DSE, so resulting solutions are not centered around a few dominant design alternatives, rather they are diversified and cover the whole Pareto-front, allowing the designer to choose the trade-off between accuracy and savings that they feel is appropriate.

Concerning the use of the AWCE on arithmetic circuits, using our methodology results in minimum-error approximate configurations exhibiting almost negligible error against savings ranging from 2% to 20%, as reported in Table 4.2, depending on error resiliency of the particular circuit being considered. Furthermore, savings provided by minimum-error approximate configurations outclasses the best of the resulting approximate configurations from [42] Thus, any comparison between the two approaches is utterly superfluous.

Figure 4.5 reports, for comparison purpose, the Pareto-fronts resulting from our methodology (represented using dots) and results from [42] (depicted as crosses), while using the AWCE metric on an 8-bits array (red), Dadda (green) and Wallace (blue) multipliers. Please kindly note that the x-axis is in semilogarithmic scale.

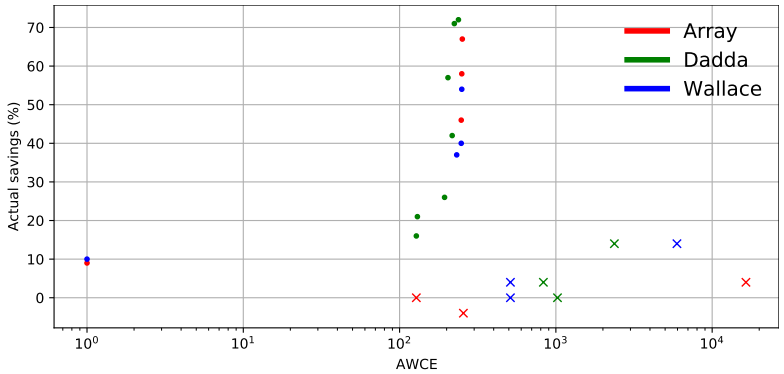



Figure 4.5: Comparison of results from [42] while using the AWCE metric on an 8-bits multipliers. Dots represent Pareto-fronts resulting from our methodology, while results from [42] are depicted as crosses. Array, Dadda and Wallace multipliers are depicted in red, green and blue, respectively.

By observing Figure 4.5, we can conclude that the use of our technique allows for the introduction of approximation into arithmetic circuits in a step-wise fashion, and that the use of a MOP-based approach for the DSE ensures that the trade-off between error and savings is carefully chosen, hence resulting in qualitatively better solutions. Indeed, using naive approximation techniques, such as stuck-at constant, and single-objective optimization approaches lead to low-end suboptimal approximate configurations.

# Chapter 5

# Image-processing case-studies

In this Chapter we discuss the application of our methodology to the design of hardware accelerators for image processing. In facts, image processing is one of the main fields of application for AxC, since imperceptible reduction of image quality can lead to important computational resources savings [47].

Section 5.2 discusses the case study of a hardware accelerator for the Sobel filter for image processing. This case study is of particular importance because the small size of the solution space allows the methodology to be compared against exhaustive exploration of the solution space. Section 5.3, instead, discusses the application of the method to the design of an accelerator for DCT, which is the most demanding step of the JPEG, viz. one of the most commonly adopted lossy image and video compression algorithm.

Before discussing the mentioned case-studies, we introduce and deeply describe the implementation of the methodology from Chapter 3 as state-of-the-art approximation framework, i.e., we present the E-𝕀DE𝔸 framework [24].

## 5.1 The E-𝕀DE𝔸 framework

Existing AxC design tools consider specific transformations and specific domains, as discussed in Section 2.3. Moreover, they are not fully automatic and simply provide a guided approach for approximation. Conversely, the E-𝕀DE𝔸 defines an automatic and general approach.

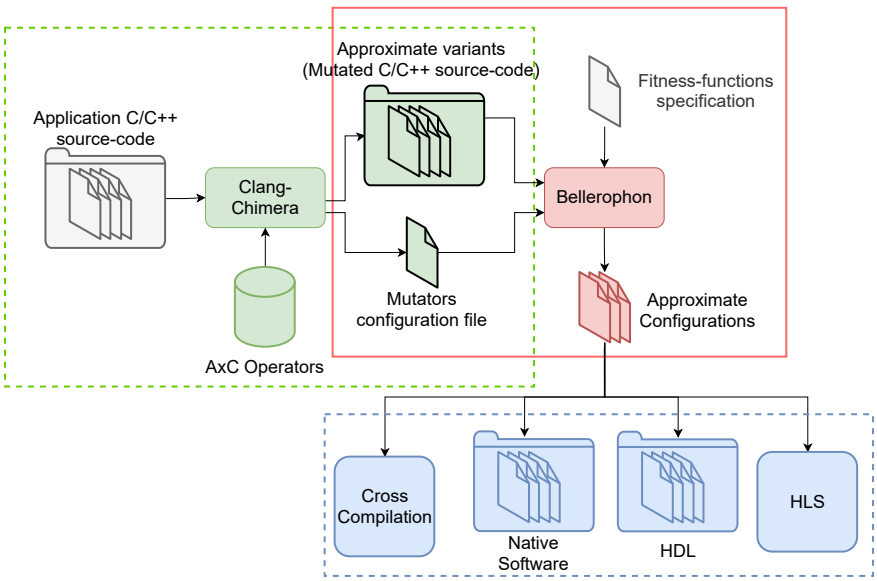



Figure 5.1: E-𝕀DE𝔸 flow, which includes Clang-Chimera and Bellerophon tool.

Figure 5.1 sketches the overall flow E-𝕀DE𝔸: it requires

1. the *original application*, described as C/C++ code;
2. the set of approximate operator, i.e., mutators;
3. the fitness-functions to select the appropriate approximation outcomes.

Concerning mutators, E-𝕀DE𝔸 is already provided with a set of mutators implementing the most common approximation techniques. For what pertains to fitness-functions, defining an appropriate error fitness-function to quantify the approximate version

deviation compared to the original outcomes is mandatory. Optionally, the user can specify an *error-threshold*, in order to determine whether the approximate outcomes are acceptable or not, and further fitness-functions, in order to quantify hardware-requirements.

As main output, E-𝕀DE𝔸 produces the mutator configuration of the best approximate application variants obtained (i.e., the non-dominated solutions). Such configurations are then used to produce software or hardware implementations, possibly involving also High Level Synthesis (HLS) tools.

E-𝕀DE𝔸 is composed of two phases carried out by two components, Clang-Chimera and Bellerophon, respectively described in Subsections 5.1.1 and 5.1.2.

### 5.1.1 Clang-Chimera

The green dotted box in Figure 5.1 reports Clang-Chimera flow. Clang-Chimera is a mutation engine for C/C++ code. It is based on the Clang compiler [3], used to rapidly develop source-to-source C/C++ compilers. Clang-Chimera applies the set of mutators (i.e., AxC Operators) to the input application code, in order to make systematic modifications to the latter, and produce a set of mutated files, i.e. the configurable approximate variants of the input application code. In addition, it also provides the *mutators configuration* file, which will be subsequently used during the DSE .

Clang-Chimera borrows the terminology from the *mutation testing* technique, which is a software testing approach used to evaluate the quality of a test set, in terms of its ability to detect software faults. Mutation technique consists in using supporting tools to mutate the original source code – thus emulating programmer mistakes – to generate erroneous programs. The test set quality is evaluated according to the number of mutated versions detected. Conversely, in the context of Clang-Chimera, the mutation is used in order to generate modified (i.e., approximate) versions of the original code. Here, the concept of software fault is replaced by the concept of approximation technique. The latter is formalized as a *mutator*.

$$Mu = \{Match, Mutate\} \tag{5.1}$$

Equation (5.1) defines the mutator $Mu$ as a pair where: **Match** identifies where in the code the mutation has to take place, defined by means of matching rules, and **Mutate** indicates how to actually modify the source code, specified by means of mutation rules.

Clang-Chimera analyzes and manipulates the input application source code through its AST, which is a tree-based representation of the application code, where each node of the tree denotes a language construct of the analyzed code. A set of AST nodes defines an AST pattern, which corresponds to a specific structure of the code. Altering the AST results in introducing constructs through which it is possible to tune the approximation degree.

Clang-Chimera utilizes Low Level Virtual Machine (LLVM)/Clang facilities, such as $ASTMatcher$ and $Rewriter$ in order to apply a given mutator $Mu_i$. In details, $ASTMatcher$ searches for all occurrences of the $Mu_i.Match$ rule and $Rewriter$ modifies the identified nodes by applying the $Mu_i.Mutate$ rule.

Clang-Chimera is already provided with a set of mutators implementing common approximation techniques, such as: (i) two loop-perforation mutators, namely LOOP1 and LOOP2; (ii) two precision-scaling mutators for the floating-point arithmetic, namely Variable Precision Arithmetic (VPA) and FLexible Arithmetic Precision (FLAP); (iii) a precision scaling mutator for the integer arithmetic, namely TRUNC; (iv) a mutator supporting approximate arithmetic operator models of circuits being part of the EvoApproxLib [127] and EvoApproxLib-Lite [129] libraries. Moreover, adding new mutators to this set allows the E-𝕀DE𝔸 framework to be easily extended. The next subsection presents some examples.

#### 5.1.1.1 Loop Perforation Example

The goal of this example is to illustrate the application of the well-known *loop perforation* technique to the C/C++ code reported in listing 5.1. The related AST is reported in Figure 5.2.

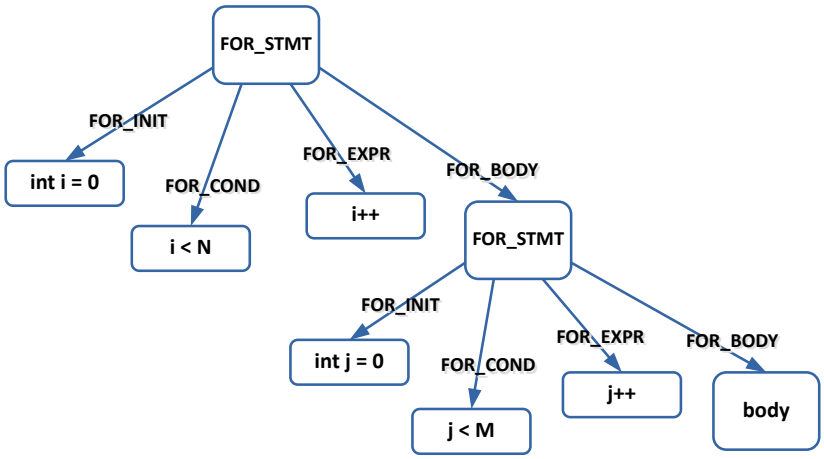



Figure 5.2: For loop AST example (see Listing5.1)

```
1 int main (void) {
2   for (int i = 0; i < N; i++) {
3       for (int j = 0; j < M; j++) {
4           body;
5         }
6     }
7 }
```

Listing 5.1: Precise Code

Basically, we want the loops to skip some iterations. This can be obtained by altering the stride of the loop variable(s). Therefore, let the Mutator related to the loop-perforation be defined as follows:

$$Mu_{LP} = \{FOR\_STMT, var{+}{+} \rightarrow var{+}{=}stride_i\} \tag{5.2}$$

The application of the mutation described in Equation (5.2) modifies the AST (see Figure 5.2). For each occurrence of $FOR\_STMT$, the corresponding $FOR\_EXPR$ node is altered. The resulting mutated code, produced by Clang-Chimera, is reported in the following listing.

```
1 int main (void) {
2     for (int i = 0; i < N; i+=stride1) {
3         for (int j = 0; j < M; j+=stride2) {
4             body;
5         }
6     }
7 }
```

Listing 5.2: Precise Code

Finally, the loop perforation effects depend on the actual values of $stride_1$ and $stride_2$ variables. Indeed, by assigning different values to the variables, different trade-offs between skipped iterations (thus performance increase) and accuracy reduction can be obtained.

#### 5.1.1.2 Approximate Circuit Example

With this example we apply mutators to alter an algorithm, such that in Listing 5.4, to be approximated by replacing every sum operation with a configurable approximated sum. The Clang-Chimera allows the automatic generation of an approximate variant having one, or more, addition operations replaced with a call to a function that implements the approximate counterpart. Let us suppose that the function performing the approximate addition is the one reported in Listing 5.3, with *add1* and *add2* being the addends and *ax* being the configurable parameters governing the approximation. The exact nature of the approximation being made by this function is hidden in the function itself: whether the function implements bit-width reduction, the latter parameter may govern the NAB of the sum; conversely, whether the function implements a library of approximate circuits, the *ax* parameter may allow selecting which particular circuit is to be adopted. Anyway, from the Clang-Chimera point of view, the particular approximate technique being implemented does not matter: using the appropriate mutator, the Clang-Chimera mutates the code in Listing 5.4 and generates the code in Listing 5.5.

As for the loop-perforation technique discussed above, the effects of approximation depend on the actual value of configuration parameters. Hence, the main problem is to find an appropriate value for these parameters, in order to achieve the best trade-off between performance gains and accuracy losses.

```
1 int ax_sum(int add1, int add2, int ax);
```

Listing 5.3: Example of approximate sum

```
1 ...
2 y = x + 2
3 z = 2 * x + 3 * y + 2;
4 ...
```

Listing 5.4: Example code to be mutated

```
1 int ax_0 = 0;
2 int ax_1 = 0;
3 int ax_2 = 0;
4 ...
5 y = ax_sum(x, 2, ax_0);
6 z = ax_sum(ax_sum(2 * x, 3 * y, ax_1),  2, ax_2);
```

Listing 5.5: Mutated code

As described in the next section, the main goal of Bellerophon is tuning these values to ultimately find non-dominated solutions in terms of trade-off between performance gain and accuracy loss.

### 5.1.2 Bellerophon

Red continuous box in Figure 5.1 depicts the Bellerophon flow. The tool analyzes the set of mutated files generated by Clang-Chimera and explores the different possible mutators configurations, i.e. different configurations for the tunable parameters in mutators, which results in different fitness values. The final result is a set of solutions corresponding to the (sub-)optimal trade-offs between the user-defined objective functions.

More formally, Bellerophon faces a MOP, which we described in Section 3.1.2.1. Given a set of decision variables, their values have to be optimized (minimize/maximize) w.r.t. the specified set of user-defined fitness-functions. Therefore, Bellerophon explores the different approximate variants while ‘moving’ towards the *Pareto front* of

the solutions, in terms of the defined fitness-functions.

Bellerophon models the MOP as follows:

1. **Population:** each solution (i.e., approximate configuration) represents an individual of the population;
2. **Chromosomes:** each individual is represented using a chromosome, as discussed in 3.1.2.2, and each of the genes correspond to an approximation parameter that can be evolved through generations;
3. **Variation Operations:** the mutation and crossover operations randomly alter genes, or combines *parents'* genes to generate an *offspring*.
4. **Fitness:** a single, or multiple user-defined fitness-functions are employed to *select* the best individuals;

Fitness-functions might be defined accordingly to the particular exploited AxC technique. In case of precision-scaling technique, for instance, a feasible fitness-function could be based on the NAB, since it translates in less hardware resources.

To perform the exploration, Bellerophon creates new populations by tuning the approximation parameters (i.e., mutator parameters in the code) and evaluates the corresponding approximate variants according to the fitness-functions, to finally select the best set. The *mutators configuration* file (see Figure 5.1) reports the maximum and minimum possible values for all approximation parameters. The main objective of Bellerophon is to converge toward optimal solutions, improving fitness-functions as much as possible.

Since implementing a full-featured NSGA-II may be cumbersome, we resorted to the ParadisEO framework [110].

In order to be evaluated, each individual has to be compiled and executed. To speed up the execution time, the compilation strategy adopted by Bellerophon allows compiling just what it is necessary to retrieve information about approximate variants. Bellerophon uses the Just-in-Time engine provided by Clang-LLVM [5]: each time the

software needs to be altered to test a new variant, Bellerophon do not invoke the system loader, rather it alters the program image which is already loaded into the memory.

The next subsection will show an example of how Bellerophon creates a population and evaluates the fitness of its individuals to then select the best set.

#### 5.1.2.1 Evolution Example

Let us refer again to the example in subsection 5.1.1.1. Clang-Chimera applied the mutations to the AST and, consequently, it generated the C/C++ code shown in Listing 5.2 and provided the "mutators configuration" file. In this example, a Mutator (i.e., approximation technique) modeling the *loop perforation* has been used. The technique has been applied to both the loops in the code (lines 2 and 3 of the code). The *mutators configuration file* produced by Clang-Chimera reports the maximum and minimum values for the two stride variable introduced (i.e., $stride_1$ and $stride_2$). The following listing reports the *mutators configuration file* generated by Clang-Chimera for this example.

Listing 5.6: Mutators configuration file example

```
stride1 , N, 1
stride2 , M, 1
```

Thanks to mutation, crossover and selection operations, Bellerophon moves towards the Pareto-front. The first population is generated randomly. Within the limits imposed by the configuration file, Bellerophon is able to mutate the individuals and generate a non-dominant population as reported in Table 5.1. In this example, $k$ individuals have

Table 5.1: Examples of a population made of $k$ individuals.

| Individual | Stride1 | Stride2 | Error | Reward |
|---|---|---|---|---|
| 0 | 3 | 5 | 5 | 8 |
| 1 | 2 | 9 | 7 | 11 |
| ⋮ | ⋱ | | ⋮ | ⋮ |
| k-1 | 5 | 2 | 3 | 7 |

been created. Each individual is characterized by its own chromosomes (i.e. value of

the stride variables), reported in the table rows. Bellerophon evaluates the individuals according to the fitness functions and assigns corresponding values to each of them. For example, here the reward is simply defined as the sum of the stride variable values: the higher the stride value, the more iterations will be skipped. However, skipping loop cycles also entails an accuracy loss. Depending on the fitness values measured for each individual, Bellerophon will select the set of the best candidates to generate the next population.

The next Section discusses a case-study targeting the design of an hardware-accelerator for the Sobel-filter edge-detection in image-processing applications. This case-study is of particular relevance since the small size of the solution space to be explored allows comparing the methodology against exhaustive exploration.

## 5.2 The Sobel-filter case-study

In this case study, we targeted the Sobel filter, which is usually employed in image processing and computer vision applications as edge-detector. As will be detailed in the following, to perform approximate variants generation and DSE, we use the E-𝕀DE𝔸 framework that we developed and presented in [24], and discussed in Section 5.1.

Aiming at reducing costs of a hardware accelerator for vertical edge detection, both in terms of silicon area and power consumption, in this experiment we configured the Clang-Chimera tool to replace exact arithmetic operators using approximate ones, by adopting implementations from the EvoApproxLib-Lite library of approximate circuits [129]. Thus, the Clang-Chimera tool generates an approximate version of the considered application in which it is possible to select, for each addition, an implementation between either the exact or an approximate implementation from the mentioned library.

Concerning optimization, during this experiment we considered three different fitness-functions, i.e. error minimization, silicon-area minimization, and power consumption minimization. We selected the PSNR as error metric, and, during error assessment, the PSNR is computed by considering images resulting from the exact and approximate Sobel filter while resorting to a comprehensive data set [6] consisting of

44 different images. Circuit area and power-consumption are estimated as the sum of the contributions of each single approximate circuit, as reported in [127].

Please, kindly note that, for this particular application, an exhaustive evaluation of the whole set of approximate configurations is computationally feasible, since the moderate amount of operations required by the filter. Indeed, vertical edge detection requires only five additions and two doubles, with the latter being implementable using wire-only left-shift, which nullifies hardware costs of multiplications. The total amount of approximate configurations is about $4.9 \times 10^7$. This gives the opportunity to compare the actual Pareto-front resulting from the exhaustive evaluation with the estimation provided by the E-𝕀DE𝔸 framework.

In order to also evaluate how the quality of the Pareto-front estimation is affected by the GA configuration parameters, we performed three different runs of the Bellerophon tool, varying the effort for the DSE phase while keeping mutation and crossover probability unmodified. In particular, (i) for the low-effort run, we considered a population of 500 individuals and 3 iterations, (ii) for the medium-effort run, we considered a population of 2000 individuals and 11 iterations, and (iii) for the high-effort run, we considered a population of 20000 individuals and 100 iterations. Table 5.2 summarizes the experimental setup.

Table 5.2: DSE parameters and relative results for the Sobel vertical edge detector case study. Note that the normalized distances is computed from fitness-function values normalized to [0,1].

| | | | | **Absolute Distance** | | | **Normalized Distance** | | |
|---|---|---|---|---|---|---|---|---|---|
| **Effort** | **Pop.** | **Iter.** | **Time** | **Min.** | **Avg** | **Max** | **Min** | **Avg** | **Max** |
| Exh. | - | - | ≈170h | - | - | - | - | - | - |
| Low | 500 | 3 | ≈5min | 0.013 | 1.58 | 7.6 | 5.9e-6 | 2.4e-4 | 1.7e-3 |
| Med. | 2000 | 11 | ≈4h | 0.002 | 1.57 | 5.8 | 3.7e-6 | 6.9e-6 | 5.6e-4 |
| Hig. | 20000 | 100 | ≈22h | 0.001 | 1.5 | 4.4 | 3.6e-6 | 6.8e-6 | 5.4e-4 |

For comparison purposes, Figure 5.3 and Figure 5.4 report, respectively, experimental results in the "PSNR vs. area" and "PSNR vs. power" perspective. Furthermore, in order to measure how close configurations $P$ provided by our methodology are w.r.t. the optimal configurations $Q$ resulting from exhaustive evaluation, we measure the distance from each point $p \in P$ to the nearest optimal configuration $q \in Q$, as

reported in Equation (5.3). The minimum, average and maximum distance are reported in Table 5.2, along with normalized distances, as done in [126].

$$d_p = \min_{\forall q \in Q} |q - p| \quad \forall p \in P \tag{5.3}$$

As the reader can figure out, results from our methodology are very close to those resulting exhaustive simulation, while, as reported in Table 5.2, the amount of time needed by exhaustive evaluation is prohibitively higher than that required by the high-effort run of the GA.

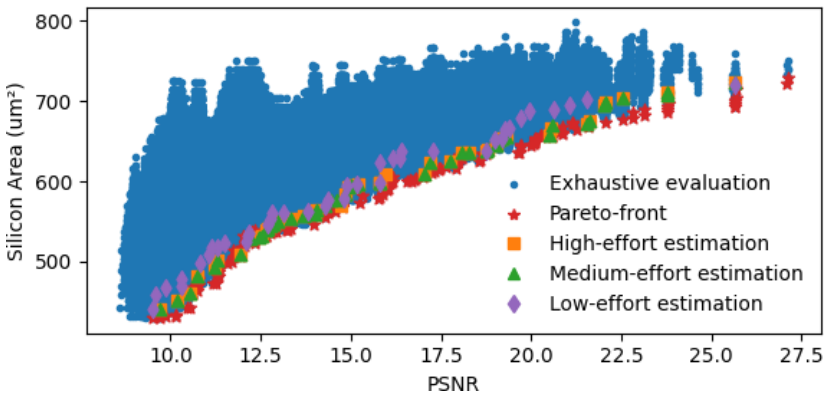



Figure 5.3: Comparison of results from exhaustive evaluation w.r.t estimation from our methodology:PSNR vs. estimated silicon area

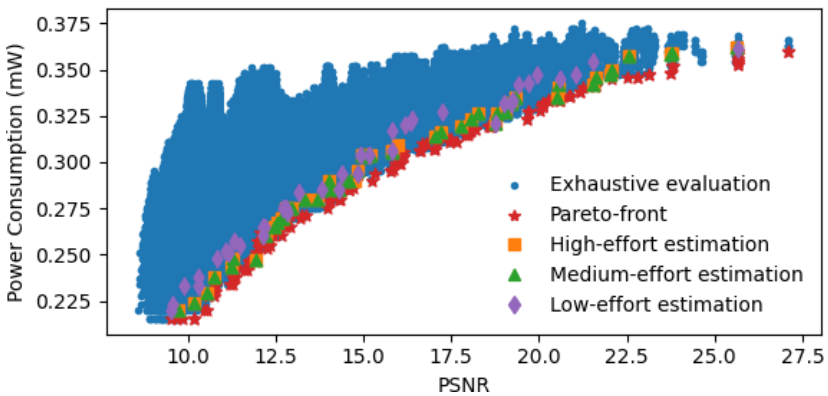



Figure 5.4: Comparison of results from exhaustive evaluation w.r.t estimation from our methodology: PSNR vs. estimated power consumption

### 5.2.1 Comparison with previous works

A comparable case study is presented in [126], where a Sobel vertical-edge detector is approximated using circuits from the EvoApproxLib8b library [127], and results from hill-climbing search, varying the effort, are confronted with those from exhaustive exploration. Our methodology is capable of providing comparable experimental setup. In particular, Table 5.3 reports comparison of results from [126] with results from our method, for comparable effort levels. As the reader can figure out, the NSGA-II allows achieving better results by at least an order of magnitude. Furthermore, when compared to the one proposed [126], our method does not require (i) the user to know how the filter is implemented, (ii) a full characterization of circuits from the library, (iii) the library pre-processing step to eliminate irrelevant circuits, which significantly reduces activities demanded to the designer, and simplifies the design process.

Table 5.3: Comparison of results from [126], in terms of normalized distance between estimated and actual Pareto-front.

| | **Results from [126]** | | | |
|---|---|---|---|---|
| **Effort** | **Avg.** | **Max.** | **Avg.** | **Max** |
| Low | 2.5e-3 | 7.5e-3 | 2.4e-4 | 1.7e-3 |
| Med. | 2.5e-4 | 1.3e-3 | 6.9e-6 | 5.6e-4 |
| High | 1e-5 | 6.5e-3 | 6.8e-6 | 5.4e-4 |

## 5.3 The DCT case-study

Most of the research work concerning image-processing applications focuses on the JPEG compression, either considering the algorithms as a whole or its individual computational steps. Concerning the design of hardware accelerators, researchers focused on the approximation of DCT accelerators, mainly targeting figures of merit such as circuit complexity, delay, area and power dissipation. Unfortunately, the effect of the different approximation techniques and relative configurations (i.e., approximation degrees) are only analyzed individually and without a supporting methodology.

In [13], for instance, a framework relying on inexact computing to perform the DCT

computation for the JPEG has been proposed. The framework acts on three levels: (i) at the application level, it exploits human insensitivity to high-frequency variation to use a filter and discard high-frequency components; (ii) at the algorithmic level, multiplier-less fast algorithms are employed for the actual DCT computation on integer coefficients; (iii) at hardware level, rather than using a simple truncation for adder circuits, authors used Inexact-Adder Cells (IACs) to compute less significant bits instead of the Full-Adder Cells (FACs). Therefore, firstly the JPEG quantization step is performed only low-frequency components of an image block; thus the high-frequency filter implementation comes down to simply setting some DCT coefficients to zero. Then, at algorithmic level, since the DCT is the most effort-demanding step in JPEG, fast DCT algorithms have been used, reducing complexity from $O(N^2)$ to $O(N)$, and requiring only integer additions. Finally, at the hardware level, different families ofIAC are considered to further reduce the power-consumption. The framework in [13] mainly aims at assessing the joint impact of those three levels of approximation. However, it presents ta rather important shortcoming, viz. approximation is introduced by manually tuning the individual approximation parameters.

Conversely, in this case study, we assess the impact of approximation on the DCT computation by performing a fully automated DSE. Applying our methodology, we start from the DCT algorithm, and we perform an AST analysis to gather information on the operations suitable for approximation. Then, we generate parametric approximate versions which allow the approximation degree to be tuned through approximation parameters. Finally, we build a MOP to find the Pareto-optimal values for the afore-mentioned approximation parameters, using the NSGA-II in order to converge towards the Pareto-front.

After providing the reader with an overview of computing the DCT using fast algorithms in Section 5.3.1, as done for all the above-discussed case-studies, we discuss the generation of approximate variants in Section 5.3.2, and several aspects concerning MOP-based DSE in Section 5.3.3, including MOP-modeling and fitness-functions to drive the DSE. Finally, in Section 5.3.4 we present experimental results.

### 5.3.1 Towards approximate DCT

As we already mentioned, the DCT computation is known to have $O(N^2)$ complexity and requires resource-intensive functional units, such as floating-point arithmetic modules. The algorithm proposed in [116] requires 11 multiplications and 29 additions to compute the one-dimensional eight-point DCT needed by the JPEG compression, and it is considered the most efficient exact algorithm, since the lower bound on the number of multiplications required for such DCT computation has been proven to be 11 [67]. In order to achieve an additional reduction in resource requirements, authors of [18] moved parts of the DCT computation to the JPEG quantization step. Furthermore, transformed coefficients can be scaled and rounded such that floating-point operations can be superseded by integer ones: the resulting algorithms are significantly faster, and they find extensive use in practical applications. However, integer multiplication is still complex and resource intensive; thus, many low-complexity multiplier-less algorithms have been proposed, such as BAS08 [26], BAS09 [27], BAS11 [28], BC12 [25], CB11 [48], PEA12 [139] and PEA14 [140]. As in [18], all of these algorithms split the DCT computation into two consecutive steps: the first one is referred to as approximate-DCT, which involves only integer operations, while the second step is embedded into the quantization and takes advantages of floating-point operations the latter requires. Moreover, they all avoid computing DCT coefficients separately or iteratively. Instead, they extensively use matrix algebra and its properties. To show how the above-mentioned algorithms work, let $X$ be an input image tile, which is a $8 \times 8$ matrix; its two-dimensional DCT transform, from now on simply DCT, is described by the following equation:

$$F = C \cdot X \cdot C', \tag{5.4}$$

where $C$ is referred to as *DCT matrix*. $C$ contains the cosine function values at the needed frequencies. The $X$ and the $F$ matrices have the same dimensions. The elements in $F$ represent the DCT coefficients as the frequency progressively increases: low-frequency components are closer to the top-left corner, while high-frequency ones are placed close to the bottom-right corner. The $C$ matrix can be split into two matrices, $T$ and $D$, as reported in Equation (5.5).

$$F = C \cdot X \cdot C' = D \cdot (T \cdot X \cdot T') \cdot D \tag{5.5}$$

Table 5.4: Comparison among DCT algorithms in terms of number of operations

| Method | Additions | Multiplications | Shifts | Total operations |
|---|---|---|---|---|
| DFT (definition) | 432 | 192 | 0 | 624 |
| FFT | 58 | 6 | 0 | 64 |
| DCT (definition) | 56 | 64 | 0 | 120 |
| Arai algorithm | 29 | 5 | 0 | 34 |
| BAS08 [26] | 18 | 0 | 2 | 20 |
| BAS09 [27] | 18 | 0 | 0 | 18 |
| BAS11 [28] (a=0) | 16 | 0 | 0 | 16 |
| BAS11 [28] (a=1) | 18 | 0 | 0 | 18 |
| BAS11 [28] (a=2) | 18 | 0 | 2 | 20 |
| CB11 [48] | 22 | 0 | 0 | 22 |
| BC12 [25] | 14 | 0 | 0 | 14 |
| PEA12 [139] | 24 | 0 | 6 | 30 |
| PEA14 [140] | 14 | 0 | 0 | 14 |

Different algorithms define $T$ and $D$ in different ways, so the number of computation operations may vary from algorithm to algorithm, as reported in Table 5.4.

Splitting $C$ allows integers-only matrix multiplications. Indeed, $T$ contains only the values $\{0, \pm\frac{1}{2}, \pm 1, \pm 2\}$ and it is orthogonal, i.e. $T' = T^{-1} \Rightarrow TT' = T'T = I$, where $I$ is the identity matrix. Note that multiplying by $\frac{1}{2}$ or 2 comes down to shifting to the right or to the left, respectively; this, at hardware level, is reduced to simple wiring. This means $T$ allows computing the DCT using only additions. Nevertheless, the multiplication of $D$ in (5.5) still requires floating-point operations. In fact, $D$ is a diagonal matrix consisting of values in the $[-1, 1]$ range, with $\{\frac{1}{2}, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{8}}\}$ being typical values. For this reason, resorting to properties of diagonal matrices allows obtaining the following equation:

$$F = T \cdot X \cdot T' \circ (diag(D) \cdot diag(D)'), \tag{5.6}$$

where $\circ$ is the Hadamard product, i.e. an element-wise multiplication. Thus, the integer null-multiplicative part $T \cdot X \cdot T'$ can be isolated from floating-point operations required by $D$. Afterwards, floating-point operations can be performed outside the DCT, and embedded into the JPEG quantization step, as shown in the following equation.

$$\begin{aligned} F_Q = \lceil F \oslash Q \rfloor &= \lceil T \cdot X \cdot T' \circ (diag(D) \cdot diag(D)') \oslash Q \rfloor \\ &= \lceil T \cdot X \cdot T' \circ \hat{Q} \rfloor = \lceil (T \cdot (T \cdot X')') \circ \hat{Q} \rfloor \end{aligned} \tag{5.7}$$

$$\hat{Q} = (diag(D) \cdot diag(D)') \oslash Q, \tag{5.8}$$

where $\hat{Q}$ in (5.8) is the *complete quantization matrix* and the $\oslash$ operator is the Hadamard division, i.e. an element-wise division. From (5.7), it follows $F = (T \cdot (T \cdot X')')$, which means that the approximate two-dimensional DCT transform can be computed using the one-dimensional DCT transform twice, reducing the complexity from quadratic to linear.

The only substantial difference between the different multiplier-less DCT algorithms is the $T$ matrix. Hence, it is straightforward to derive a set of equations to calculate the single dimensional DCT coefficients. Equations in (5.9), for instance, refers to the BC12 algorithm [25].

$$\begin{aligned} f_0 &= x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 & f_1 &= x_0 - x_7 \\ f_2 &= x_0 - x_1 - x_2 + x_3 + x_4 - x_5 - x_6 + x_7 & f_3 &= x_4 - x_3 \\ f_4 &= x_0 - x_3 - x_4 + x_7 & f_5 &= x_5 - x_2 \\ f_6 &= x_2 - x_1 + x_5 - x_6 & f_7 &= x_6 - x_1 \end{aligned} \tag{5.9}$$

Furthermore, some terms – for instance $(x_0 + x_7)$ – are involved in the computation of multiple $f_i$ coefficients, which allows to further reduce the amount of operations.

### 5.3.2 Generating of approximate variants

Once the addition-based equations for the DCT coefficients are defined, simple implementations for the DCT computation algorithm can be derived. Within those, we introduce further approximation by replacing exact sums by configurable approximate ones. Such approximate sums allow setting two parameters, i.e., the NAB and the type of adder cell to use (namely, a classic FAC or an IAC). We take into account three different IAC families, i.e., the Approximate Mirror Adder (AMA) [75], the Approximate XOR-based Adder (AXA) [183] and the IneXact Adder (InXA) [12].

As mentioned, this is the same approach adopted in [13]. However, while in [13] the approximation was manually introduced, we automate the replacement process by considering the AST of the algorithm implementation, resorting to E-𝕀DE𝔸 [24].

Thus, we model each of the above-mentioned multiplier-less DCT algorithms

by using C/C++ implementations straightly derived from equations (5.7) and (5.9), and starting from such implementations, the generation of approximate variants is performed using the Clang-Chimera tool [24]. For each DCT algorithm, the tool produces mutated sources which allow configuring, for each of the sums, both the NABs and type of adder hardware cell to use (i.e., either FAC or IAC).

### 5.3.3 Design-space exploration

Decision variables of the MOP are parameters introduced during the approximate variants generation step, i.e., the NAB value and the type of adder hardware cell to be used for each of the approximate operations. Thus, if $N_{op}$ is the number of addition required by a given algorithm, each approximate configuration is identified through the use of a vector, i.e., a chromosome, which is composed of $2 \cdot N_{op}$ different elements, viz. genes. Chromosomes are provided with an additional gene representing the approximation degree for the high-frequency filter. Thus, each chromosome is composed of $2 \cdot N_{op} + 1$ genes.

Concerning the error fitness-function, we resort to the SSIM [175] to evaluate differences among images. Its formal definition is reported in Equation (5.10), where $X$ and $Y$ are two sets of data (i.e., the images), $\mu_X$ and $\mu_Y$ are their mean values, $\sigma_X^2$ and $\sigma_Y^2$ are their variances, $\sigma_{XY}$ is their co-variance, $L$ is the value range in which elements of X and Y can vary, and $k_1$ and $k_2$ are tuning parameters (typically equal to 0.01 and 0.03, respectively). Values of $SSIM(X, Y)$ span in the range $[-1, 1]$. Values of $SSIM(X, Y) \approx 1$ mean that $X$ and $Y$ are structurally similar, while values of $SSIM(X, Y) \approx 0$ mean that there is no similarity between the two images. Values smaller than zero are meaningless [175].

$$SSIM(X, Y) = \frac{(2\mu_x\mu_y + k_1) \cdot (2\sigma_{xy} + L \cdot k_2)}{(\mu_x + \mu_y + k_1) \cdot (\sigma_x^2 + \sigma_y^2 + L \cdot k_2)} \tag{5.10}$$

Since, in practice, a single overall quality measure of the entire image is required, the Mean SSIM (MSSIM) from Equation (5.11) is adopted. There, $X$ and $Y$ are the reference and the distorted images, respectively, $x_j$ and $y_j$ are the image contents at the j-th *local window*; and $M$ is the number of local windows in the image. Typically, the MSSIM index is computed considering $11 \times 11$ Gaussian weighted circular windows

rather than on $8 \times 8$ square tiles [175].

$$MSSIM(X,Y) = \frac{1}{M}\sum_{j=1}^{M} SSIM(x_j, y_j) \tag{5.11}$$

As SSIM, the lower the MSSIM index the lower the similarity between X and Y sets; thus, in order to define a suitable fitness-function for the MOEA to minimize error, we adopt the Structural DISSIMilarity (DSSIM) – $DSSIM(X,Y) = 1 - MSSIM(X,Y)$. In particular, we compute the DSSIM between a standard JPEG compressed image $X$ and an image $Y$ which is obtained by using a certain approximate configuration of a given approximate algorithm. Both $X$ and $Y$ originate from the same non-compressed source image. We perform this operation for several images and use the average DSSIM as the final error fitness-function. For what pertains to error assessment, we resort to the whole JPEG compression, performed on a representative data set, to estimate the error. The considered data set [6], consists of 44 different images, covering a wide set of common features, including among others a flat gray scale, foreground subject with a messy background, and high contrast images.

Concerning silicon-area requirements, we again resort to model-based estimation to drive the DSE. In particular, we estimate the silicon-area from the number of transistors required to implement an inaccurate cell, using the data from [13]. For convenience, in Table 5.5 we report, from [13], the number of transistors required to implement inaccurate cells of the mentioned IACs.

Table 5.5: Transistor count from [13] for inexact-adder cells

| Cell | Full Adder | AMA1 | AMA2 | AMA3 | AMA4 | AXA1 | AXA2 | AXA3 | InXA1 | InXA2 | InXA3 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Transistors | 58 | 20 | 14 | 11 | 14 | 8 | 6 | 8 | 6 | 8 | 6 |

Let us detail the reward function. Let $N_{op}$ be the number of operations required to compute the single-dimensional DCT and let $nab_i$ be the NAB for the *i*-th addition. We compute the total number of saved transistors as

$$\sum_{i=0}^{N_{op}-1} nab_i \cdot (T_{FA} - T_{IAi}), \tag{5.12}$$

where $T_{FA}$ and $T_{IAi}$ are the number of transistors required by the FAC and the *i*-th IAC,

respectively. Finally, since the number of additions required by each algorithm varies, we use a normalized measure, as reported in the following equation:

$$\rho = \frac{1}{2 \cdot N_{bits} \cdot N_{op} \cdot T_{FA}} \sum_{i=0}^{N_{op}-1} nab_i \cdot (T_{FA} - T_{IAi}) \tag{5.13}$$

where $N_{bits}$ and $N_{op}$ are the number of bits on which each of the sums is expressed and the number of sums required for the DCT computation, respectively.

### 5.3.4 Evaluation and experimental results

In this section, we firstly describe the DCT algorithm hardware implementation, then we show our experimental setup and related results, and, finally, we perform a comparison with previous work.

#### 5.3.4.1 Hardware implementation

In order to be able to measure the final gains, we encoded all the above-mentioned DCT algorithms in VHDL. Such implementations guarantee high flexibility: they handle the configuration of both the type of adder cells to use for each addition and the number of bits to approximate (NABs). This allows the synthesis of any solution eventually found in the DSE process. VHDL implementations follow Equation (5.7), which allows splitting the two-dimensional DCT into two consecutive one-dimensional DCTs, separated by a transposition block, which transposes the signals. The transposition block implementation in hardware comes down to being just wiring. A block schema of the two consecutive one-dimensional DCTs is depicted in Figure5.5: $X_1, \cdots, X_7$ represent the rows of the image tile being transformed, while $F_1, \cdots, F_7$ represent the rows of the transformed block. A RTL schema of the single dimensional DCT computation block (DCT1D) is shown in Figure 5.6; without loss of generality, the schema refers to BC12 [25], since the differences between different algorithms are negligible. The architecture of the one-dimensional DCT computing block is pipelined, with pipe registers separating the adders needed for the partial-sums computation. The one-dimensional DCT has three clock cycles latency, thus the whole two-dimensional DCT

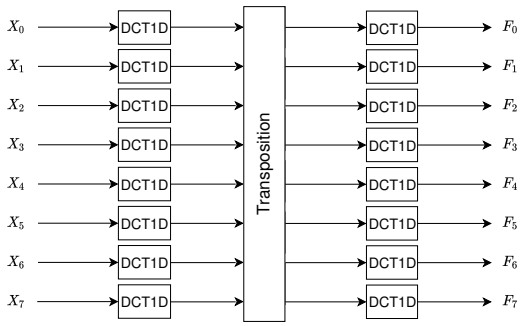



Figure 5.5: RTL block schema for the BC12-2D hardware implementation

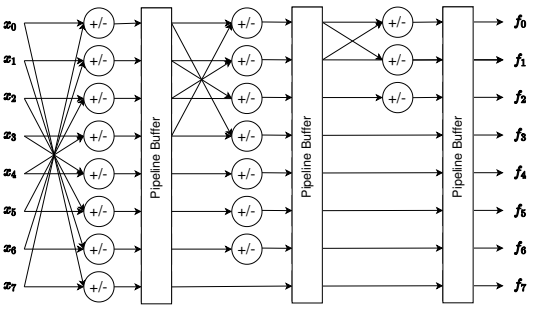



Figure 5.6: RTL block schema for the BC12-1D hardware implementation

block is six clock cycles latency. Each of the partial sums is performed using a configurable approximate adder. The scheme of a configurable approximate adder is depicted in Figure 5.7: it is a ripple-carry adder whose least significant bits are computed by IACs, while the most significant ones are computed by classical FACs. The number of approximate sums, i.e. IACs, is configurable by means of the NAB parameter. The DCT is computed on $8 \times 8$ image tiles, each one made of three different color

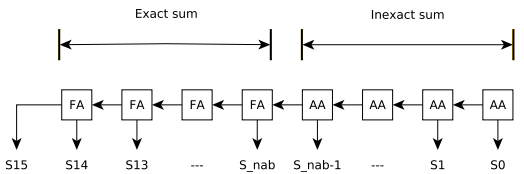



Figure 5.7: Inexact ripple-carry adder

channels. Each element's value spans from 0 to 255. For this reason, each one of the single-dimensional DCT output terms can be expressed, at most, as the sum of eight elements. Therefore, the maximum value for the single-dimensional DCT terms is $8 \times 255 = 2040 < 2048 = 2^{11}$. As a consequence, the two-dimensional DCT output terms can have a maximum value of $8 \times 2040 = 16320 < 16384 = 2^{14}$. As a result, 14 bits turn out to be sufficient to represent the DCT frequency coefficients. It is also worth highlighting that replacing FACs with IACs leaves the overall structure unchanged.

#### 5.3.4.2 Experimental results

As we mentioned above, seven different DCT algorithms and ten types of IACs are considered during this case study. As for the DCT algorithms, we considered BAS08 [26], BAS09 [27], BAS11 [28], BC12 [25], CB11 [48], PEA12 [139] and PEA14 [140]. As for the IACs families, we considered AMA [75], AXA [183] and InXA [12]. All of these are encoded in the C++ language, and the generation of approximate variants is performed exploiting the Clang-Chimera tool we discussed in Section 5.1.1. The latter variants are, then, evolved leveragind the Bellerophon tool.

During DSE we set the Bellerophon tool to use an initial population equals to 2000 individuals, mutation and crossover probabilities set to 0.7 and 0.9, respectively, and three generations as stop-criterion. We did not set any maximum error threshold. It is worth noting that exhaustive DSE is undoubtedly unfeasible, even in the case evaluating a single solution requires negligible time, since the size of solution spaces ranges between $2.66 \times 10^{49} \approx 2^{164}$ and $1.081 \times 10^{80} \approx 2^{265}$

Figure 5.8 reports the Pareto-front provided by the NSGA-II for all the considered algorithms. The reference is, for each algorithm, its non-approximate implementation, depicted as a gold star. It is important to bear in mind that such algorithms are non-exact DCT versions (see Section 5.3.1) and that the JPEG implemented with a non-exact DCT algorithm produces lower-quality images compared to its exact version. For this reason, the reported non-approximate solutions exhibit already some error. Their reward value is zero, since they do not use any IACs, so they do not achieve any approximation gain according to Equation 5.13. As envisioned, the graphs highlight increasing expected

rewards as the error increases.

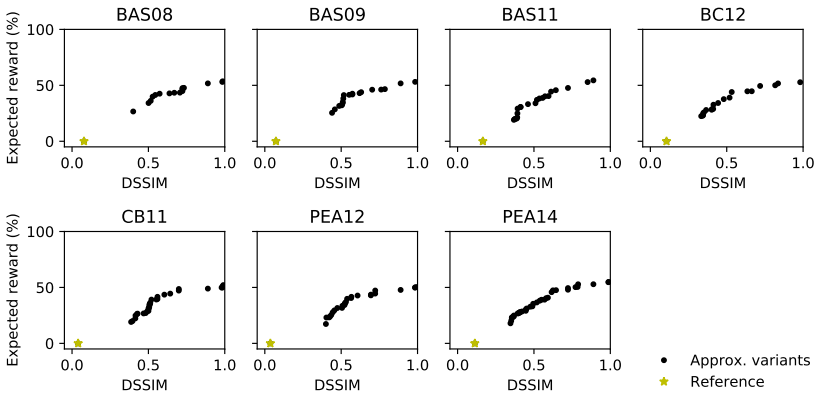



Figure 5.8: Pareto-front estimation provided by the MOEA

After the DSE, to correctly evaluate the final gains, we synthesized the obtained approximate configurations to both ASIC and FPGA technologies. Over all the algorithms, the total number of obtained non-dominated approximate configurations to synthesize was 164, i.e., $\approx 24$ per each algorithm, on average. For the reader convenience, in the following figures we plotted the experimental result data along with the corresponding first-order interpolation to highlight the trend.

**5.3.4.2.1 ASIC Synthesis** We synthesized all the obtained non-dominated approximate configurations to ASIC, by using the 65nm FinFET technology and the Cadence *Genus Synthesis Solution* tool. We resorted to the synthesis reports for the silicon-die area of the approximate configurations. In Figure 5.9, we report the result.

Concerning the power consumption, to determine whether the synthesis power report provides a satisfying accuracy, we simulated the whole workload for two algorithms (BAS08 and BAS09) and collected the resulting power consumption. As a result, we realized that the difference between the power consumption resulted from the workload simulation and that estimated by the synthesis tool only differed by 5%, on average. We considered the synthesis report accuracy sufficient, thus in Figure 5.10 we show the power results from the synthesis report. Please note that the scale on the left axis (static power) is different from the scale on the right axis (dynamic power).

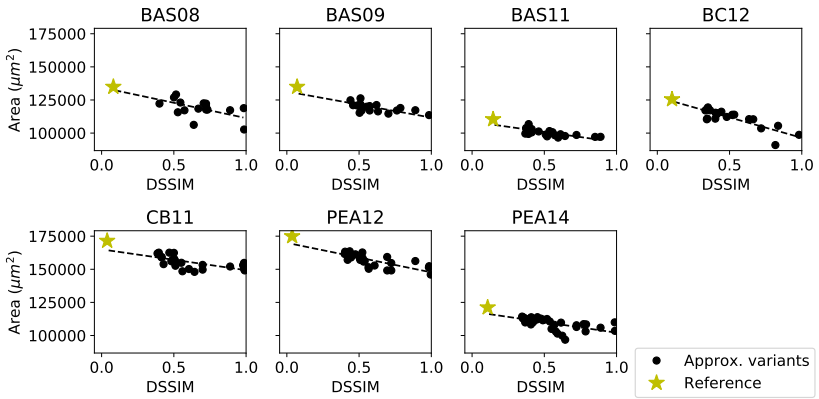



Figure 5.9: silicon-die area requirements (in $\mu m^2$)DCT hardware-resource requirements while targeting the 65nm FinFET technology

Power savings are achieved due to both the reduced area and the lower switching activity that IACs exhibit w.r.t FACs, as also reported in [13]. It is worth highlighting that the trends shown in Figures 5.9 and 5.10 are perfectly in line with the trend predicted by our approach (see Figure 5.8). Indeed, higher reward in Figure 5.8 corresponds to lower area/power in Figures 5.9 and 5.10. For the reader convenience, Table 5.6 reports a summary of the minimum and maximum area/power savings we achieved during the experimental campaign while targeting ASIC.

Table 5.6: Minimum and maximum savings while targeting ASIC

| Algorithm | Area Savings (%) | | Power Savings (%) | |
|---|---|---|---|---|
| | min | max | min | max |
| **BAS08** | 9 | 25 | 5 | 20 |
| **BAS09** | 5 | 15 | 5 | 15 |
| **BAS11** | 5 | 12 | 9 | 13 |
| **BC12** | 6 | 27 | 6 | 25 |
| **CB11** | 5 | 17 | 3 | 10 |
| **PEA12** | 7 | 17 | 5 | 15 |
| **PEA14** | 5 | 23 | 4 | 18 |

**5.3.4.2.2 FPGA Synthesis** We synthesized all the obtained non-dominated approximate configurations to a Xilinx Zynq-7020 MPSoC. To get a fair estimation of hardware requirements, we used only its embedded FPGA and inhibited DSPs usage. Figure 5.11 reports synthesis result in terms of number of LUTs for all the considered algorithms. As expected, approximate solutions require less resources than the precise implemen-

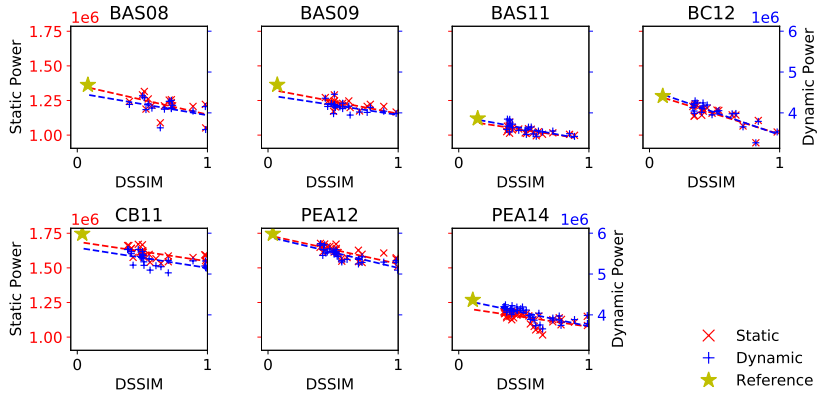



Figure 5.10: Power-consumption requirements (in nW) DCT hardware-resource requirements while targeting the 65nm FinFET technology

tation, as highlighted by the decreasing general trend. In order to correctly evaluate energy savings, we performed a post-synthesis timing simulation, using the Dynamic Power Analysis tool provided by the Xilinx Vivado. In this case, since the synthesis report has a very low confidence level for power consumption estimation, we resorted to a workload simulation for all the solutions the DSE provided, for all the algorithms. In this way, we achieved a high confidence level power estimation.

Figure 5.12 shows static and dynamic power consumption for all the algorithms. The static power of the FPGA is largely caused by the fabric of the device and does not directly depend on used resources, while dynamic one is directly linked to the user design, due to the input data pattern and the design internal activity. Being our hardware implementations of approximate DCT characterized by low overhead, i.e. device resources usage falls between 6 and 13 %, it is necessary to split power consumption in static and dynamic since the former turned out to be about an order of magnitude greater than the latter one for the target FPGA device.

Also in this case, power savings are achieved thanks to both the reduced total area and the logical structure of IACs: FPGA LUTs implementing IACs have a lower switching activity than those implementing FACs, as reported in [13].

As in the ASIC case, also for FPGA the trends shown in Figures 5.11 and 5.12 are perfectly in line with the trend predicted by our approach (see Figure 5.8). Indeed,

higher reward in Figure 5.8 corresponds to lower area in Figures 5.11.

As done for ASIC, we report a summary of the minimum and maximum area/power savings we achieved during the experimental campaign while targeting FPGA in Table 5.7.

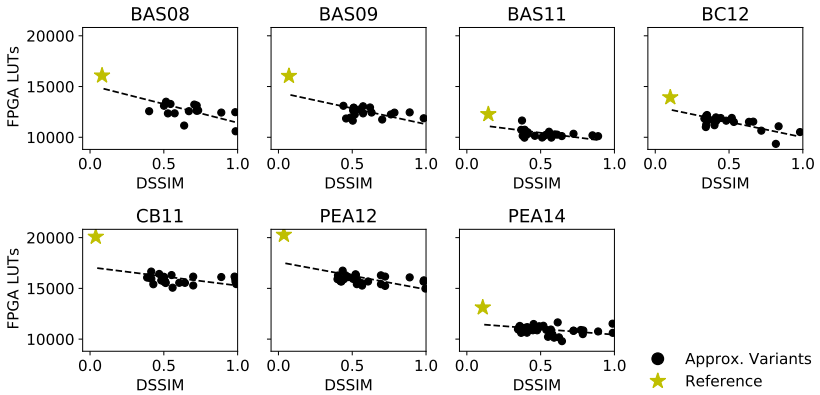



Figure 5.11: LUTs requirements while targeting a Xilinx Zynq-7020 FPGA

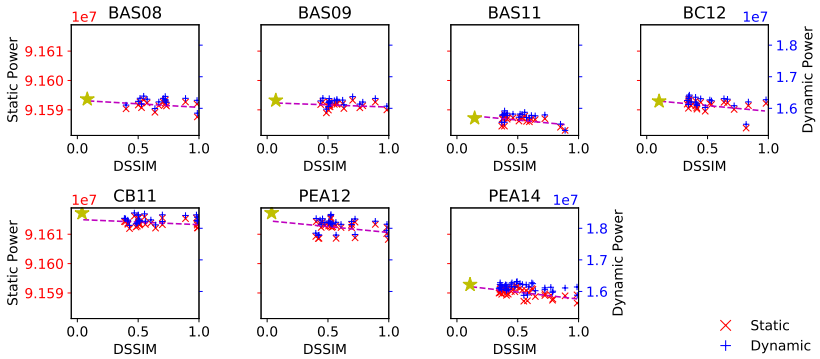


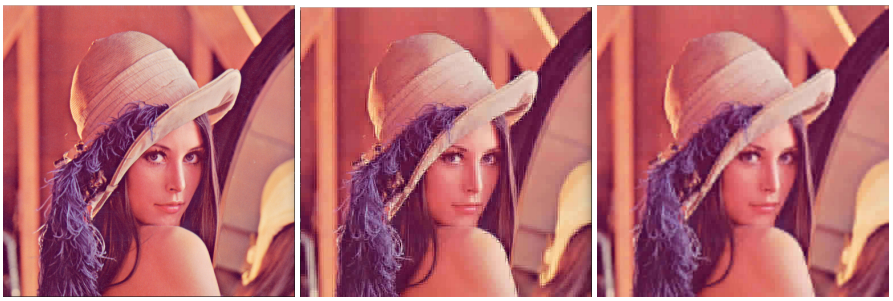

Figure 5.12: Power-consumption (in nW) while targeting a Xilinx Zynq-7020 FPGA

**5.3.4.2.3 Visual Test** Since JPEG belongs to image processing domain, we also provide a visual test: Figure 5.13 shows, from left to right, the standard JPEG-compressed image of Lena and Baboon, the same images compressed using the exact version of the BC12 algorithm [25] – which exhibit a DSSIM of 0.10, and requires 125473.92 $\mu m^2$ and 5691946 $\mu W$ when implemented on ASIC, or 5902 LUTs and 107933980 $\mu W$

Table 5.7: Minimum and maximum savings for FPGA synthesized approximate configurations

| Algorithm | LUTs Savings (%) | | Dynamic Power Savings (%) | |
|---|---|---|---|---|
| | min | max | min | max |
| **BAS08** | 27.5 | 48.2 | 0.1 | 3.4 |
| **BAS09** | 32.6 | 42.5 | 0.3 | 1.8 |
| **BAS11** | 30.6 | 40.4 | 0 | 2.5 |
| **BC12** | 31.1 | 50.8 | 0 | 5.2 |
| **CB11** | 30.9 | 42.7 | 0.5 | 2.6 |
| **PEA12** | 31.3 | 42.7 | 0.7 | 4.4 |
| **PEA14** | 29.2 | 44.5 | 0.1 | 2.8 |

while targeting FPGA – and, finally, the ones compressed with its approximate variant having 0.33 as DSSIM value and 0.22 of reward, which correspond to 8362.64 $\mu m^2$ and $352.711\mu W$ saved for ASIC and 1846 LUTs and $94506.744\mu W$ saved for FPGA. As the reader can easily figure out, the quality differences are barely perceivable.



(a) Visual test with Lena

(b) Visual test with Baboon

Figure 5.13: Visual test

#### 5.3.4.3 Comparison with previous work

In this subsection, we compare the results obtained with our approach with those obtained in the work in [13]. Authors of [13] estimated gains $G$ through the following equations:

$$G = \frac{V_i - V_e}{V_e} \tag{5.14}$$

$$V_i = P_i \cdot nab + (N - nab) \cdot P_e \tag{5.15}$$

where $V_i$ and $V_e$ represent the average energy required to perform an addition, by an inexact N-bits adder and by an exact N-bits adder respectively, $P_i$ and $P_e$ represent the average energy required by a single IAC and by a full-adder cell, respectively. Values of $P_i$ and $P_e$ used in [13] were measured by using the 45nm Complementary Metal-Oxide Semiconductor (CMOS) technology and are reported in Table 5.8. Such

Table 5.8: Energy consumed by a single adder cell from [13]

| Cell | | FullAdd | AMA1 | AMA2 | AMA3 | AMA4 | AXA | InXA1 | InXA2 | InXA3 |
|---|---|---|---|---|---|---|---|---|---|---|
| Energy(fJ) | Avg. | 0.9267 | 0.513 | 0.6631 | 0.6649 | 0.478 | 0.4042 | 0.1535 | 0.0563 | 0.3409 |
| | Max. | 2.3668 | 0.9794 | 0.7203 | 0.7116 | 0.6271 | 0.8924 | 0.2096 | 0.1291 | 0.4211 |

equations have the same goal as Equation (5.13), i.e. predicting the gains achieved thanks to the approximation. While Equations (5.14) and (5.15) take into account the energy consumption parameters of the individual adder cell, Equation (5.13) takes into account only the number of transistors.

In [13], authors performed a manual exploration. In particular, firstly they tried different IACs and decided to always resort to InXA2 in their experiments, based on its energy delay product. Then, they tried different NAB values for the InXA2 adder and finally set it to 4 for all the experiments (i.e., for all the DCT algorithms). Besides, they used the PSNR metric to measure the JPEG error entailed by the approximate DCT variants. Conversely, we adopted the DSSIM index as error metric – which is more suitable for image processing – and we let the MOEA decide which inaccurate cell to use and how many bits to approximate (i.e., the NAB parameter) for each of the sums. A minor difference concerns the implementation of the adders: while 32-bit adders were considered in [13], we considered 14-bit adders.

In order to effectively compare the two studies it is necessary to place them under the same conditions. Thus, we executed the JPEG algorithm on the same four images considered in [13] – i.e., Lena, Cameraman, Boat and Pepper – by using the approximate DCT variants obtained with our approach and computed the PSNR metric. Hence, we computed energy savings according to Equation (5.14), considering 32-bit adders. Figure 5.14 shows the obtained results. Concerning both energy consumption

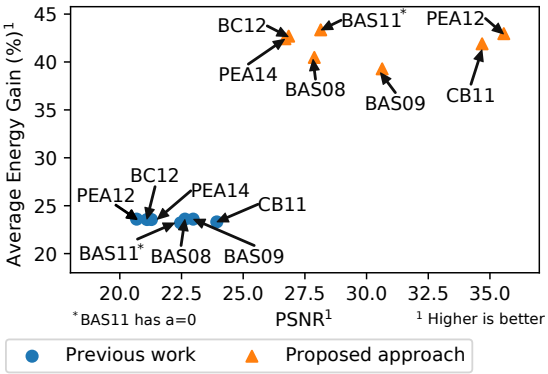



Figure 5.14: Comparison with results from [13]

and PSNR, our approach allowed a significant improvement for all the considered algorithms compared to the approach adopted in [13]. In detail, our approach allowed an absolute improvement spanning from 15.69% to 20.15% (average 18.38%) concerning the energy gain and from 5.24 dB to 14.88 dB (average 7.91 dB) concerning the PSNR. Therefore, with our approach we were able to produce higher quality images, i.e. with less error, while consuming less energy. This is the result of the thorough DSE made possible by the proposed approach. Indeed, using a MOEA allows performing a multi-objective optimization more efficiently and automatically. Moreover, not needing to synthesize each approximate variant allows exploring more extensively the design space in a reduced time.

# Chapter 6

# Artificial intelligence case-studies

This Chapter discusses the application of our methodology on two of the most promising classification models in the machine-learning domain, namely DNNs and DT MCSs.

As we discuss in the following, state-of-the-art hardware accelerators targeting DNNs and DT MCSs are quite resource intensive, due to the massive amount of processing elements needed to effectively accelerate computations. In either case, the high demands in terms of both silicon area and power consumption utterly hinder the spread of commercial devices. However, performance of computing systems can often be enhanced by exploiting inexact computation, as discussed in Section 2.1. Indeed, the AxC design paradigm is effective in a wide range of error-resilient applications, including data analytic, scientific computing, multimedia and signal processing, and machine-learning [47].

The desire to reduce the hardware overhead of accelerators, however, cannot jeopardize more than half a century of efforts to achieve the accuracy that modern models exhibit. In facts, keeping a reasonably high accuracy level while introducing approximation to save resources are conflicting design goals; hence, besides challenges inherently arising while exploiting the AxC paradigm, we must cope with the difficulty

of obtaining substantial resource benefits while sacrificing a minimal, ideally negligible, amount of classification accuracy.

Imposing error constraints while optimizing for silicon area and/or power consumption in a single-objective optimization fashion typically results in solutions which are in the neighborhood of local optimums [56]. Conversely, out MOP-based DSE is able to provide a full Pareto-front consisting of several trade-offs between considered fitness-functions to optimize.

For the mentioned models, in order to provide the reader with the required knowledge to understand the challenges that accelerating these models poses to the designer, in addition to a brief history and a few important literature references, the training methodology by which the models can be trained to solve a particular problem is covered, the state of the art in hardware accelerator architectures is presented, and how the AxC paradigm can be leveraged to reduce the cost of these accelerators is discussed.

## 6.1 Neural Networks

In recent years, Artificial Neural Network (ANN) have won numerous contests in pattern-recognition, classification, object-recognition and so forth, imposing themselves on everyone's attention as one of the most successful learning technique. However, the concepts behind them have been circulating since the early 1940s [155], although early models do not learn, they consisted of only a single processing element, and their design were based on linear-regression method. This is why these early models are referred to as linear classifiers.

The first multi-layer stack of simple modules consisting of fully-interconnected processing elements, each computing non-linear input-output mappings, and some other embryonic concepts concerning supervised-learning were introduced in the late 1960s, as part of the Group Method of Data Handling (GMDH) family of inductive algorithms [91, 90]. The observation leading to GMDH was that linear classifiers can only carve their input space into very simple regions, namely half-spaces separated by a hyperplane. Conversely, more complex problems require the input-output mapping

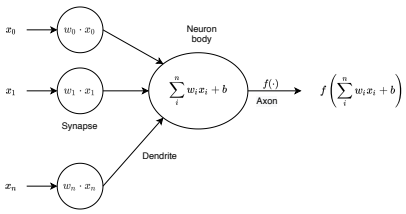
to be not sensitive to irrelevant variations of the input, while being very sensitive to particular minute ones [106].

Neuroscience-related concepts, which influenced the research concerning DNNs, also dates back to 1960. The computational model of artificial neurons, for instance, is inspired by its biological counterpart: neurons receive stimulus from *dendrites* and produce an output along the *axon*, which eventually branch, and connect to other neurons through *synapses*. However, this simplified model is quite different from that of biological neurons. Indeed, biological dendrites do not simply carry signals but, as well as biological synapses, they perform very complex – and still partially unknown – non-linear functions [155].

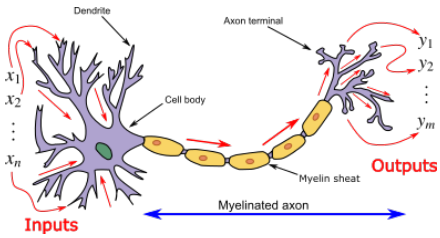
The most commonly adopted computational model of artificial neurons is depicted in Figure 6.1a, along with its biological counterpart. Input signals are multiplied with learned weights at synapses, while dendrites carry the weighted input-signals to the neuron body, where partial-products are summed and biased. An output is produced along the axon – i.e., the neuron "fires" – if the weighted sum is greater than a threshold, defined by the neuron's *activation function*. Another substantial difference between the biological and computational model resides in the exact instant in which the neuron "fires" that encodes the information, not the frequency of firing. Some more details will be discussed in Section 6.1.1, while activation function will be briefly discussed in Section 6.1.2.

Also Convolutional Layers (CLs) and Pooling Layers (PLs) are directly inspired by the classic notions of simple-cells and complex-cells of visual neuroscience, and the overall architecture of Convolutional Neural Networks (CNNs) is reminiscent of the hierarchy of cells in the visual cortex of cats [88]. We For now, we postpone presenting the layered architecture of modern DNNs, since they are discussed in Section 6.1.3.

CLs and PLs were embedded in a learning system for the first time only in the early 1980s, as part of the Neocognitron system [69], along with several other concepts, such as weight-sharing, which became common in later systems. Although the Neocognitron was the first system that truly deserved the "deep" attribute, and despite the mentioned innovations, the actual learning were performed using a non-supervised winner-take-all approach, rather than the well-known backpropagation algorithm. This is because the latter algorithm were developed independently by different research group only in late




(a) Computational model of a neuron.




(b) Biological model of a neuron, from [70].

1980s, albeit the error-minimization approach through gradient descend was discussed since early 1960s in [63]. Nevertheless, backpropagation were initially performed by computing standard Jacobian matrices, without taking into account neither direct links between interconnected layers nor sparsity [155].

The first embryo of the backpropagation algorithm dates back to [113], albeit it was not related to DNN. However, it was soon explicitly adopted to minimize the cost function by adapting weights during the DNNs training phase [177, 104]. The algorithm will be briefly discussed in Section 6.1.1.

However, for a while, the backpropagation algorithm seemed to work only for swallow networks. The reason is that typical DNNs suffer from the famous problem of vanishing or exploding gradient: cumulative back-propagated error either shrink rapidly or grow out of bound [83]. This is known as the fundamental problem in deep-learning, or long-time-lag problem, and, over the years, several ways of partially overcoming it were explored, including (i) very deep-learners, which alleviate the problem using unsupervised pre-training, (ii) long short-term memory Recurrent Neural Networks (RNNs), which adopt *constant error carousels*, i.e., units using the identity function as activation function, and constant unitary weight in feedback connections [85], (iii) weight search without relying on gradient descent, such as random weight guessing [84], (iv) the introduction of Max-Pooling Layers (MPLs) for subsampling [176], and (v) back-propagating the error further layers down. The latter was essentially made possible by the adoption of GP-GPUs to speed up the backpropagation algorithm [155].

CNNs were soon combined with MPLs, alternating CLs and MPLs [148]. However, the real breakthrough happened when Max-Pooling Convolutional Neural-Networks (MP-CNNs) were trained using standard GP-GPU-accelerated backpropagation, orig-

inating the structure of many modern competition-winning feed-forward deep learners [155]. In facts, in 2010, the 0.4% error record on the Modified National Institute of Standards and Technology (MNIST) database of handwritten digits [104] was broken by an ANN trained using input-distortion and GP-GPU-accelerated backpropagation [53]. Only one year later, the record was broken again by a MP-CNN trained with standard GP-GPU-accelerated backpropagation [52]. The latter result suggested that advances in modern computing hardware were more important than advances in learning algorithms [155].

Despite these successes, CNNs were largely forsaken by the mainstream computer-vision and machine-learning communities, until an ensemble of MP-CNNs achieved super-human performances in visual pattern recognition, during the International Joint Conference on Neural Networks (IJCNN) 2011 traffic-signs competition [50]. This result is particularly interesting since it paved the way for self-driving cars. The very same year, another MP-CNNs achieved super-human performances, setting a new record of 0.2% on the MNIST database of handwritten digits [49]. Also, the ImageNet object-detection competition was won in 2012 by a MP-CNNs. With respect to previous contests, the ImageNet significantly raised the challenge, involving $256 \times 256$ pixels images, far larger than $32 \times 32$ or $48 \times 48$ of previous contests. The winning MP-CNN achieved spectacular results, halving the error rates of the best competing approaches [102]. This was particularly relevant also because object-detection has numerous industrial and medical applications, including surveillance and biomedical diagnosis [51].

These successes came from the efficient use of GP-GPU, the Rectifier Linera Unit (ReLU) activation function (which will be detailed in Section 6.1.2), a new regularization technique called dropout [162], and techniques to generate more training examples by deforming the existing ones. This success has brought about a revolution in computer vision, and CNNs are now the dominant approach for almost all recognition and detection tasks [106].

### 6.1.1 The backpropagation algorithm

In order to describe the backpropagation algorithm, we resort to the compact event-oriented notation for activations spreading in DNNs from [155], which is simple and general enough to accommodate both feed-forward DNNs and RNNs.

Assume $i, j, k, t, p, q, r$ positive integer variables with range from the given context, and $m, n, T$ positive integer constants. Although the topology of an DNN can vary over time, it can be described using a set $N = \{u_1, u_2, \cdots\}$ of units, i.e., neurons, and a set $H = N \times N$ of directed edges, or connections, between neurons. As we mentioned, instead of being modeled as an amorphous blob of interconnected elements, the latter are organized in layers, with edges connecting only elements belonging to adjacent layers.

The firs layer, the one receiving raw input-data, is, precisely, the input layer. In feed-forward DNNs, the k-th layer is the set $L_k \subset N$ of neurons such that there is an edge-path length of at most $k-1$ between an input node and $u \in L_k$. Anyway, there may be shortcuts such that the edge-path length is less than $k-1$, but no longer path between input neurons and neurons belonging to the k-th layer. This also implies no cycle is allowed in feed-forward DNN.

The behavior of the network, at runtime, is governed by a set of real-valued learned weights $w_i$, $i = 1 \cdots n$. During an *episode*, i.e., during information processing and activation spreading in the network, there is a partially causal sequence $x_t$, $t = 1 \cdots T$ of real values called *events*, which can be either *input-event* or the activation of a unit $u \in N$ that directly depends on $x_k$, k < t, through causal connections. Let the $\upsilon$ function encodes the network topology by mapping the $(k, t)$ pair to weights. During an episode, the same weights may get reused over and over again, in a topology-dependent manner, encoded by the $\upsilon$ function. This is called weight-sharing and, as mentioned, it is exploited to decrease the descriptive complexity of CNNs and RNNs, by reducing the amount of learned weights.

For non-input events, we may have

$$x_t = f\left(net_t\right) \tag{6.1}$$

where $f$ is typically a non-linear function, i.e. the activation function, and $net_t$ can be either

$$net_t = \sum_{k \in in_t} x_k \cdot w_{v(k,t)} \tag{6.2}$$

or

$$net_t = \prod_{k \in in_t} x_k \cdot w_{v(k,t)} \tag{6.3}$$

In supervised learning, some $x_t$ events may be associated with teacher-given values $d_t$, yielding error

$$e_t = \frac{1}{2}(x_t - d_t)^2 \tag{6.4}$$

The goal of supervised learning in network training is to find weights that yield episodes with small error $E = \sum_t e_t$ in the hope that the network will generalize well in later episodes, causing only small error when fed with previously-unknown input-events.

The backpropagation algorithm to compute the gradient of an objective-function w.r.t. learned weights of a multi-layer stack of modules is nothing more than a practical application of chain-rules for derivatives. The key insight is that the derivative of the objective functions w.r.t. inputs of a module can be computed by working backward from the gradient computed w.r.t. outputs of that module, or the input of the subsequent module. The backpropagation equations can be applied repeatedly, to propagate gradients through all modules, starting from the output module all the way back to the input module. Once these gradients have been computed, computing the gradient w.r.t weights is straightforward.

The chain rule of derivatives tell us how small effects are composed. Consider a small change $\Delta x$ in $x$, which gets transformed into a small change $\Delta y$ of $y$ by getting multiplied by the gradient of $y = f(x)$, i.e., by $\frac{\partial y}{\partial x}$, hence

$$\Delta y = \frac{\partial y}{\partial x} \Delta x \tag{6.5}$$

Similarly, a small change $\Delta y$ in $y$ creates a change $\Delta z$ in $z$ by getting multiplied with

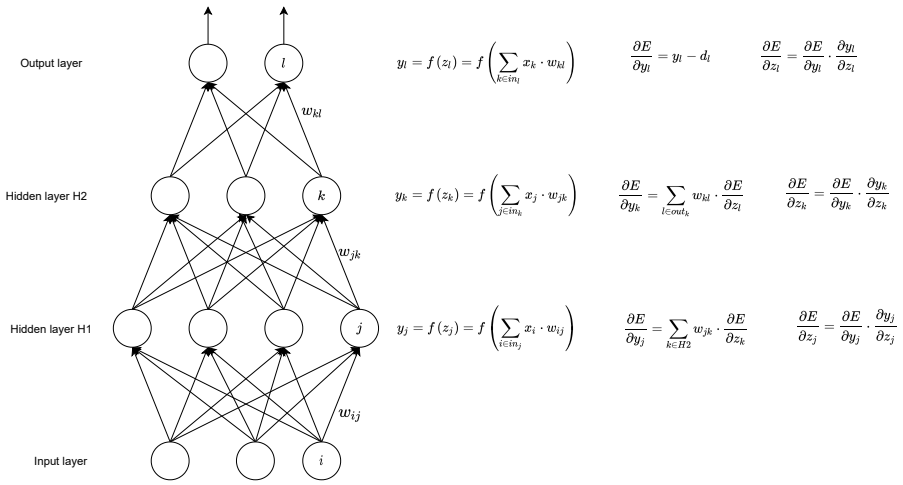



Figure 6.2: Example of backpropagation

$\frac{\partial z}{\partial y}$, thus

$$\Delta z = \frac{\partial z}{\partial y} \Delta y \tag{6.6}$$

By substituting Equation (6.5) in Equation (6.6), i.e., by applying chain-rules for derivative, we obtain a description of how $\Delta x$ may affect $\Delta z$

$$\Delta z = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \Delta x \tag{6.7}$$

Equation (6.7) also describe how the gradient $\frac{\partial z}{\partial x}$ can be computed, i.e.

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \tag{6.8}$$

Chain rules for derivative work also when $x$, $y$, and $z$ are vectors. Consider, for instance, the network depicted in Figure 6.2, which has two hidden layers, each constituting a module through which one can back propagate the gradient. During the forward-pass, at each layer, the total input $z$ is first computed as the weighted sum of the output of neurons belonging to the previous layer; then, the activation function $f(z)$ is computed to get the final output.

During the backward-pass, instead, at each hidden layer the error-derivative w.r.t. the output of each layer is computed. This is the weighted sum of the error-derivative w.r.t the total inputs to units in the subsequent layer. For the node $k$ of Figure 6.2, for instance, it is

$$\frac{\partial E}{\partial y_k} = \sum_{l \in out_k} w_{kl} \cdot \frac{\partial E}{\partial z_l} \tag{6.9}$$

Then, the error derivative w.r.t. the input $\frac{\partial E}{\partial z_k}$ at node $k$ can be computed by multiplying the derivative of $f(z) = \frac{\partial y_k}{\partial z_k}$ by the error-derivative w.r.t. the output from Equation (6.9), hence

$$\frac{\partial E}{\partial z_k} = \frac{\partial E}{\partial y_k} \cdot \frac{\partial y_k}{\partial z_k} = \frac{\partial y_k}{\partial z_k} \cdot \sum_{l \in out_k} w_{kl} \cdot \frac{\partial E}{\partial z_l} \tag{6.10}$$

Once the latter is known, the error-derivative for weights $w_{jk}$ on connection from the neuron $j$ to the previous layer is just

$$y_j \frac{\partial E}{\partial z_k} \tag{6.11}$$

Therefore, in order to minimize the error, weights have to be updated by following the opposite direction w.r.t the computed gradient.

### 6.1.2 Activation functions

As we mentioned, neurons compute non-linear mapping between inputs and outputs. On purpose, suitable activation-functions are suitably adopted. There are several activation functions: the *sigmoid* (6.12), for instance, was largely used in the past because it takes real-valued inputs and squashes it to the [0,1] range. However, it has some drawbacks: it rapidly saturates, nullifying the gradient during stochastic gradient-descend algorithm, and its output is not zero-centered, which is utterly less severe than saturation, though it introduces some undesirable dynamics during the stochastic gradient-descend algorithm.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{6.12}$$

The *tanh* function (6.13) is a scaled sigmoid that squashes input in the [-1, 1] range, therefore output is zero-centered. Anyway, the gradient nullification still persists.

$$tanh(x) = 2\sigma(2x) - 1 \tag{6.13}$$

The ReLU (6.14) has become very popular in the last few years. It simply thresholds to zero, and it was found to greatly accelerate the stochastic gradient descent algorithm w.r.t the sigmoid and the *tanh*, due to its linear and non-saturating form. Unfortunately, ReLU units can "die" during training and their death is not reversible, since training data could cause the weights to update in such a way that the neuron will never activate on any data point again.

$$relu(x) = \max\{0, x\} \tag{6.14}$$

The dying neuron issue can be addressed either using a proper learning rate or using the leaky-ReLU activation function (6.15). The latter employs the $\alpha$ parameter – which is typically very small – to allow the activation function to have a small negative slope. In this way, the gradient will be less prone to be zero.

$$lrelu(x) = 1(x < 0) \cdot \alpha \cdot x + 1(x \geq 0) \cdot x \tag{6.15}$$

A different approach to overcome the mentioned issue involves the adoption of the *maxout* activation, that offers the benefits of the ReLU – i.e. linearity – yet does suffer neither from dying neurons nor from saturation. Nevertheless, it increases the amount of parameters of the network.

$$maxout(x) = \max_i\{W_i' \cdot x + b_i\} \tag{6.16}$$

### 6.1.3 Layers in neural networks

As we mentioned, instead of being modeled as an amorphous blob of connected neurons, DNNs are organized in distinct *layers*. The number of layers defines the network's depth. Neurons belonging to adjacent layers can be either fully or partially connected.

In Fully-Connected Layers (FCLs), neurons of the layer are connected to all of the preceding layer, while in CLs, neurons in a layer are connected only to a small region of the previous layer. In any case, there is no connection between neurons within the same layer. In RNNs, cycles are allowed, while they are strictly forbidden in feed-forward ANNs, such as CNNs. Moreover, CNN architecture is constrained to be arranged in three dimensions, and explicit assumptions are made on the input, which is images, unveiling a more efficient forward function implementation, and a reduction in the amount of learned parameters.

Three main types of layers are peculiar to CNNs, namely the CL, the PL and the FCL. While PLs perform a single function, i.e., sub-sampling, CLs and FCLs perform computation that are not just function of the inputs, but also of learned synaptic-weights and biases. PLs are placed in between CLs, to progressively reduce the spatial size of the intermediate representation. Anyway, it is possible to define parametric CLs so that the input-volume reduction is performed during the convolution [80]. Thus, PLs are not always necessary in the networks.

A FCL takes a 3D-input vector, namely a *tensor*, $X \in \mathbb{R}^{H_{in} \times W_{in}}$, and multiplies it with a kernel $K \in \mathbb{R}^{W_{in} \times W_{out}}$, producing the output tensor $Y \in \mathbb{R}^{H_{in} \times W_{out}}$ according to the following equation:

$$Y_{i,j} = \sum_{k}^{W_{in}} X_{i,k} \cdot K_{k,j} + b \tag{6.17}$$

A CL takes a tensor $X \in \mathbb{R}^{H_{in} \times W_{in} \times C_{in}}$ as input, it multiplies that tensor with a tensor $K \in \mathbb{R}^{H_k \times W_k \times C_{in} \times C_{out}}$, and produces $Y \in \mathbb{R}^{H_{out} \times W_{out} \times C_{out}}$, according to the following:

$$Y_{k,l,n} = \sum_{i}^{H_k} \sum_{j}^{W_k} \sum_{m}^{C_{in}} K_{i,j,m,n} \cdot X_{k+i-1,l+j-1,m} + b_m \tag{6.18}$$

where $H$, $W$, and $C$ are, respectively, the height, the width and the number of channels (depth) of each involved tensor.

Since the amount of computation required by modern DNNs, several proposals concerning accelerating such computations have been proposed in the scientific literature.

Such include using either GP-GPUs or custom hardware targeting the ASIC or FPGA technology. In the following Section, we provide the reader with a brief summary on this topic.

### 6.1.4 Accelerators targeting Neural Networks

Hardware solutions for the development and deployment of ANNs are quite diversified, and include either general-purpose solutions such as CPUs and GP-GPUs, FPGAs and even special-purpose ASICs. It is not trivial to recognize which of these solutions is the best one, since a substantial amount of design constraints could lead to a choice for one or the other solution.

In the scientific literature, architectures of ANN accelerators are often classified in two main classes, i.e. temporal and spatial. As for the former, PEs are Arithmetic Logic Units (ALUs) with no memory capacity and centralized control units. As for the latter, PEs are typically complex and interconnected units with internal memory and their own control units [37].

CPUs and GP-GPUs are examples of temporal architectures. The former are the least used for training and inference, since their throughput is limited by the small number of cores and the small amount of operations executable in parallel. A number of attempts to tackle with ANNs using CPUs have been made, including the Intel AVX-512VNNI [92], the bfloat16 reduced bit-width data representation format [35], and the *bigDL* software library of optimized algorithms for distributed CPU clusters [55]. The thousands of cores to work efficiently on parallel algorithms make GP-GPUs the most widely adopted solution for accelerating neural networks. In particular, for the training phase. Indeed, the most popular deep-learning frameworks – including *TensorFlow*, *PyTorch* and *Caffe* – support GP-GPUs, and vendors typically provide highly-optimized and GP-GPUs-accelerated software libraries for common layers. An example is as the NVIDIA *cuDNN* library [45]. Furthermore, recent GP-GPUs from NVIDIA also provide tensor-cores [134], which are optimized to perform large matrix operations, mixed-precision multiply-and-accumulate operations, and also exploit sparsity for an additional performance boost.

Accelerators implemented on FPGAs and ASICs are typically categorized as spatial architectures, and they are not only distinguished by more complex and interconnected PEs, but also for an accurately designed memory hierarchy. Indeed, DRAM accesses are the actual bottleneck for computation, since DRAM has a higher latency and higher energy requirements w.r.t. smaller and faster SRAM adopted for caching; therefore, many research works focused on data reuse either inputs, weights and biases. Accelerators from [64, 40] perform partial-sum reuse, while inputs and weights are distributed to PEs according to the portion of the input activation map being processed. In [72, 96], synaptic weights and biases are reused, while inputs are distributed to PEs in order to also reuse partial-sums. Finally, the approach from [43, 44] aims at maximizing the reuse of all data, by mapping a row of the convolution to the same PE, which keeps fixed weights.

In order to further improve performances and energy efficiency, several other strategies than data reuse have been proposed, such as sparsity exploitation, quantization, and reconfiguration.

Opportunities to exploit sparsity originate from both the weights redundancy and the widespread use of the ReLU activation function (See Section 6.1.2). As for the former, it is possible to prune – i.e. nullify – synaptic weights [77], while the use of the ReLU activation function nullifies all negative values within activation maps. Both phenomena can be effectively exploited to avoid multiplications whose result is already known, and to compress activation maps or weights, consequently reducing the memory overhead of the model.

A variety of solutions exploiting sparsity by combining different compression schemes have been proposed in the scientific literature. Early approaches either compress weights or activations. In [9], for instance, activation maps are compressed, but sparsity of weights is not considered. Instead, authors of [188] compress weights, but do not consider sparsity of activation maps. Anyway, both activation and weights compression are considered in recent works [73, 137]. Authors of [81] go even further, exploiting repetitions of any value rather than only null weights. This allows reusing partial products, partial sums, and also paves the way for reducing the memory footprint of the model. Nevertheless, compression is not the only available solution: zero-skipping for null activation is effectively exploited in a number of different accelerators, in order to lower the energy consumption [76, 99, 8, 109, 44]. In particular,

[109] leverages concise convolution rules to avoid performing operations resulting in null values, while [44] also perform computations on compressed data in order to avoid wasting energy in data decompression.

One of the techniques to overcome memory bottleneck is bit-width reduction: rather than using the IEEE 754 32-bit floating-point arithmetic for both training and inference, the latter phase can be efficiently performed using fixed-point or even integer only arithmetic, with a negligible classification-accuracy loss, by exploiting quantization [93]. In order to take advantage of quantization, not only in terms of memory but also for energy consumption, several variable bit-width accelerators have been developed [97, 157, 158, 150, 108]. Almost all of them adopt a serial approach for the multiply-and-accumulate part of the computation, mitigating the increased latency by exploiting inherent parallelism of computations to be performed within a layer.

Concerning flexibility and configurability, the increasing interest in deep-learning lead to the development of a large variety of models and layers exhibiting very different features. Though, almost all the accelerators are designed and optimized targeting a specific model. Nevertheless, to allow for more widespread deployment of ASIC and/or FPGA accelerators, flexible and reconfigurable design are desired. Flexible data flows for exploiting data reuse in CLs have been proposed in [117, 167], while dedicated architectures for both CLs and FCLs have been developed in [78]. Reconfigurable accelerators, such as [103, 141], exploit Network on Chip (NoC) reconfiguration to adapt the data flow to different layers.

### 6.1.5 Approximate DNNs

DNN requires intensive computations both during the training and the inference phase. Concerning the latter phase, hardware accelerators exhibit substantial hardware requirements, due to the many processing elements to perform computations. Each processing element has multiple arithmetic operations, such as multiplications, accumulations and activation functions, which definitely consume area and power. The multiplication, in particular, is recognized as the most demanding operation both in terms of silicon area and power consumption, therefore, several optimizations have been proposed in the scientific literature, including using binary weights, so multiplications collapse in

logic-AND [54], or weights in the form of powers of two, so that multiplications can be performed as left-shifts [112].

As we mentioned, the inner error-resiliency of ANN makes them the ideal field of application for AxC; consequently, a significant amount of research focused on both the training and the inference phase, attempting to further reduce resource requirements of hardware accelerators. In the following, we briefly report some of the most relevant contributions.

One of the approach to identify approximation-resilient neurons in ANNs is that discussed in [174]. It leverages the backpropagation algorithm, which has been briefly discussed in Section 6.1.1, to obtain a measure of the sensitivity of the output of the considered DNNs, to the output of each neuron. Indeed, backpropagation redistributes the error at the DNN outputs backward all the way to its inputs, thus, it quantifies the error contribution of each neuron towards the global error. This allows to identify neurons that contribute the least to the global error, i.e., neurons that are amenable to approximation. The latter are sorted based on the magnitude of their average error contribution, and a whether the latter falls below a predetermined threshold, they are labeled as resilient or sensitive. Resilient neurons are replaced using approximate ones, which are designed using the precision-scaling technique, and allow modulating the bit-widths of both inputs and weights on the basis of resilience. A subsequent retraining step suitably adjusts the learned parameters, alleviating the impact of approximation-induced errors, and allowing further approximation. A similar approach has been proposed in [187], in which error-resiliency is exploited to implement memory-access skipping techniques.

Albeit it focuses on approximating synapses, rather than neurons, the work in [100] also leverages the backpropagation algorithm to characterize the error resiliency. According to authors, approximating synapses works on a finer grain w.r.t. approximating neurons, since approximating neurons implies approximating all synaptic weights the same way. The precision-scaling technique is adopted to modulate the bit-width of synaptic weights, while a greedy approach is used to determine a suitable precision, in order to minimize the power consumption while keeping acceptable accuracy. In addition, they also employ approximate multipliers in order to further reduce the power consumption. Working with such a fine grain, however, requires coping with several issues, including (i) moving data exhibiting different approximation degree back and

forth from memory, and (ii) the possibility that non-approximate weights may constitute the input of approximate multipliers.

As mentioned, multipliers are recognized as the most demanding component within neurons. Therefore, as foreseeable, several contributions focus precisely on them.

Authors of [17] investigated on properties an approximate multiplier should exhibit in order to maintain acceptable classification accuracy and, at the same time, reduce the use of silicon area. They observed that low values for the variance $\sigma_{ED}$ and Root Mean Squared Error (RMSE), which are reported in Equation (6.20) and Equation (6.19), respectively, make the multiplier not deteriorate the classification accuracy. Furthermore, they noted that whether a multiplier underestimate or overestimate the exact product with equal probability, the classification accuracy tend to increase, since such a multiplier prevents the errors from accumulating. However, this is a necessary-but-not-sufficient condition.

$$RMSE = \sqrt{E\left[e^2\right]} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} e_i^2} \tag{6.19}$$

$$\sigma_e = E\left[e - E\left[e\right]\right] = \frac{1}{N}\sum_{i=1}^{N}\left(e_i - \frac{1}{N}\sum_{i=1}^{N} e_i\right)^2 \tag{6.20}$$

In the same work, the authors also investigated the impact of reduced-precision multipliers on classification-accuracy, observing that using 8-bit multipliers causes negligible degradation in classification accuracy, and that the smallest multiplier design to provide an acceptable classification accuracy without requiring network retraining is the 6-bit multiplier. On the other hand, network retraining allows using 4-bit multipliers with only 2% degradation in classification accuracy.

In order to alleviate approximation-induced error, the mentioned contributions from the scientific literature resort to network-retraining. While, on one hand, this even allow for further approximation, on the other hand it increases the overall design time. Moreover, retraining is not even possible, e.g., the dataset on which the network has been trained may not be available. A way to overcome this limitation while using approximate multipliers has been proposed in [130]: a weight-tuning algorithm adapts the learned weights to the employed multipliers, allowing accuracy recovering. The

proposed algorithm exploits the fact that, for each of the multiplications, the value of the one operand – the one holding the synaptic weight – is constant, while the second operand varies with the input data. Thus, a *map-function* can be computed offline, and exploited during approximation to determine the suitable weight-update.

In [128], the EvoApprox8b library of arithmetic components [127], designed through CPG as in [156], is further evolved, and employed to conduct a resiliency analysis targeting CNNs, specifically, different networks belonging to the ResNet [80] family. In order to preemptively lower resource requirements, 8-bits quantization is exploited. Further savings are pursued using approximate hardware components, selected as in [126]. In this regard, the EvoApprox8b is expanded considering both standard $n \times n$ and $m \times n$ approximate multipliers, and CNNs are approximate either considering a single layer at a time or the network as a whole. As for the former, all exact multipliers of a single layer are replaced using approximate multipliers selected from the library. As for the former, all multiplications of all layers are replaced using one particular implementation taken from the mentioned library, regardless of layers' resiliency.

Reviewing the reported contributions, the following drawbacks can be easily recognized: (i) they are related to a specific approximation technique; (ii) they typically require a re-training step to alleviate the impact of approximation on the classification accuracy, which undoubtedly increases the design time; (iii) they either optimize a single parameter (silicon area, for instance) under quality constraints, or combine multiple design objectives in a weighted single-objective optimization problem; therefore, the resulting solutions may be centered around a few dominant design alternatives and do not explore the whole Pareto-front [56], and (iv) although they recognize that each neuron may contribute to error and hardware-requirements differently, depending on the layer it belongs to, the degree of introduced approximation does not take into account these differences.

Conversely, as extensively discussed in Chapter 3, our methodology (i) supports different approximation techniques; (ii) does neither leverage the backpropagation algorithm nor require retraining, so as to avoid lengthening the design time and make the method applicable even when retraining is not possible; (iii) allows for the different degree of error resilience exhibited by different parts of the same application to be taken into account; and (iv) is based on multi-objective optimization, which allows

for solutions that simultaneously optimize multiple figures of merit, e.g., classification accuracy, silicon-area or LUTs, and power-consumption.

### 6.1.6 Applying the methodology to DNNs applications

In this Section we discuss several case-studies concerning the application of our methodology to DNNs, along with the hardware accelerator we designed to assess hardware requirements.

As will be detailed in the following, to perform approximate variants generation and DSE, we use the E-𝕀DE𝔸 framework that we extensively discussed in Section 5.1. Hence, through a MOP-based DSE, we find the correct approximation degrees leading to non-dominated solutions exhibiting near-Pareto trade-offs between accuracy-loss and hardware-efficiency. Indeed, E-𝕀DE𝔸 allows specifying multiple fitness-functions; for our scenario, we define the accuracy-loss and the hardware requirements to be both minimized. Furthermore, our approach allows to selectively introduce approximation within layers while considering the network as a whole, contextually analyzing error resiliency of layers.

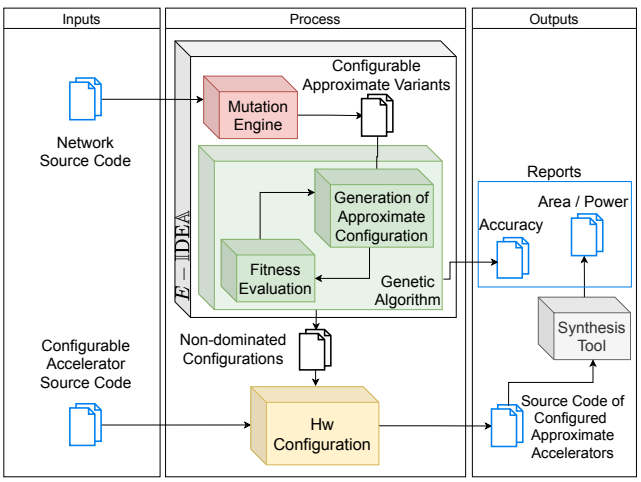



Figure 6.3: Actual workflow for DNNs approximation.

Figure 6.3 sketches our workflow as a whole: given a DNN implementation, the

first step is approximate variant generation, which is performed by exploiting the Clang-Chimera tool. From a technical standpoint, we consider C/C++ implementations of DNNs, designed and trained by making use of the Neural Network Design & Deployment (N2D2) framework [114], which is an open-source CAD framework for DNNs design and simulation.

The approximate variants resulting from Clang-Chimera are fed to the Bellerophon DSE engine: approximate configurations resulting from assigning a value to each configuration parameter within variants are evaluated in terms of fitness-functions, and Pareto-optimal ones are selected using the NSGA-II heuristic. At the end of the DSE, the latter configurations are exploited to configure the accelerator we expressly designed in order to assess the actual hardware requirements through FPGA synthesis and simulations. The architecture of such accelerator will be discussed in Section 6.1.6.3.

#### 6.1.6.1 Generating approximate variants

Since the target application – i.e., DNNs – is extensively researched, taking into account the results presented in the scientific literature is helpful beyond any doubt while identifying approximable parts and suitable approximation technique. Indeed, as discussed in Section 6.1.5, the scientific literature identifies multipliers within neurons as the most demanding component in terms of resource requirements. Therefore, they can be considered as good candidates a fortiori. The scientific literature also provide suitable approximation techniques to be adopted, depending on the particular final implementation being considered. As we discussed above, albeit many other techniques may be taken into consideration, including the loop-perforation, the memoization and the load-value approximation, the precision-scaling and inexact mathematical operators are the most commonly adopted ones.

The precision-scaling technique, for instance, can be applied by carefully manipulating the AST in order to supersede precise multiplications and/or additions in (6.17) or (6.18) with approximate ones. As discussed, such approximate operations should allow selecting the appropriate degree of approximation to be introduced, by means of tunable parameters. Nevertheless, the amount of such parameters can easily explode, since the amount of operations within layers. Structural properties of layers, how-

ever, can be exploited in order to reduce the amount of introduced parameters while effectively introducing approximation. In CLs, for instance, *weights-sharing*, which reduces the amount of parameters to be learned during the training phase by sharing synaptic weights among neurons within the same layer, allows applying the same approximation degree to all neurons belonging to the same CL. However, operations in different CLs must have their own approximation degree. Neurons belonging to FCLs usually do not share synaptic weights, yet they process the same input volume. Thus, neurons belonging to the same FCL can share the same approximation degree. PLs can also be subject to approximation. Their contribution in terms of calculation burden is, however, negligible w.r.t. CLs and FCLs. Moreover, sub-sampling by means of stride in convolutions is progressively supplanting PLs. Therefore, we do not apply any approximation to such kind of layers.

We configured Clang-Chimera to truncate input operands and results of multiplications in CLs and FCLs. Thus, the Clang-Chimera tool produces an approximate version which allows configuring, for each of the approximate layers, the approximation degree for the multiplications and additions involved in the weighted sum (6.17) or (6.18), depending on the considered layer.

#### 6.1.6.2 Design space exploration

In order to estimate the error introduced by the approximation, we configured Bellerophon to execute the approximate CNN on the training test data set, in order to assess the classification-accuracy loss. Concerning hardware-requirements, we estimate savings by taking into account several parameters that definitely have some impact on the former, such as: (i) the input-volume of a neuron, which impacts the amount of operations performed within it; (ii) the amount of neurons within a layer, i.e., the output volume size of a layer, which impacts the hardware requirements of the whole layer. Nevertheless, the definition of the actual fitness-functions for estimating hardware-requirements must take into account the adopted approximate technique, and its impact on requirements themselves. Therefore, for each of the case study we report in the following, we discuss a suitable reward function.

#### 6.1.6.3 Configurable hardware architecture

In order to evaluate resource requirements and savings resulting from our methodology, we designed and implemented a configurable accelerator suitable to be used for FPGA synthesis.

The RTL block schema of our accelerator is depicted in Figure 6.4: PEs (the blue spheres in Figure 6.4) are organized as two-dimensional grid of $r$ rows and $c$ columns, with $r$ and $c$ being user-configurable to accommodate different data-reuse necessities. Indeed, PEs on the same row process the same input volume (red cubes of Figure 6.4), applying different synaptic weights (green cubes in Figure 6.4), thus computing a portion of a fiber of the output activation map. On the other hand, PEs on the same column apply the same synaptic weights to different input volumes, hence computing a portion of the corresponding channel of the output feature map (white "spaghetti" in Figure 6.4). Therefore, the amount of rows and columns of the grid controls the amount of fibers and/or channels of the output features map being simultaneously computed, respectively. Whether a fiber is computed along the whole depth depends on the number of columns of the grid, while whether an output channel is computed along both width and height depends on both the amount of rows of the grid and how input data are fed to the accelerator. Besides CLs, FCLs can also be implemented, by using a single-row grid. A RTL block schema of PEs is depicted in Figure 6.5.

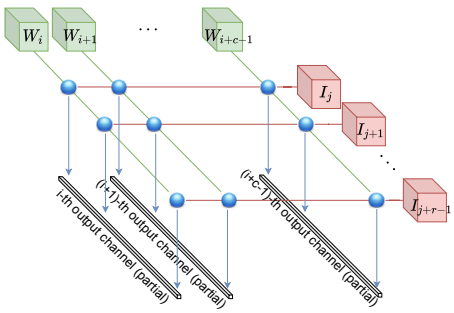



Figure 6.4: RTL block schema of our DNN accelerator

Essentially, one PE implements a whole neuron, and takes advantage from the massive inner-parallelism hardware provides, performing unrelated operations by exploiting parallel multipliers and adders. Inputs and weights signals are fed into a multiplier

block (depicted in light gray) that computes partial-products using $d \times h \times w$ parallel multipliers, where $d$, $h$ and $w$ are, respectively, the depth, height and width of neurons' input-volume. Partial-products are, then, fed into a binary-tree based sum-reduction block (depicted in light blue), that consists of $log_2\left(d \times h \times w\right) + 1$ levels each of which consists of $2^l$ parallel adders, with $l \in [0, log_2\left(d \times h \times w\right)]$ and $l = 0$ being the root-node of the reduction tree, i.e. the one computing the final sum. Then, the activation function is computed on the final sum, in order to compute the final output. PEs are pipelined, with multipliers consisting of three pipe stages, the sum-reduction block consisting of $log_2\left(d \times h \times w\right) + 1$ pipe stages and two further stages are within the activation function block. Hence, the latency of PEs is $log_2\left(d \times h \times w\right) + 6$ clock cycles. In order to guarantee high flexibility, PEs handle the configuration of (i) the

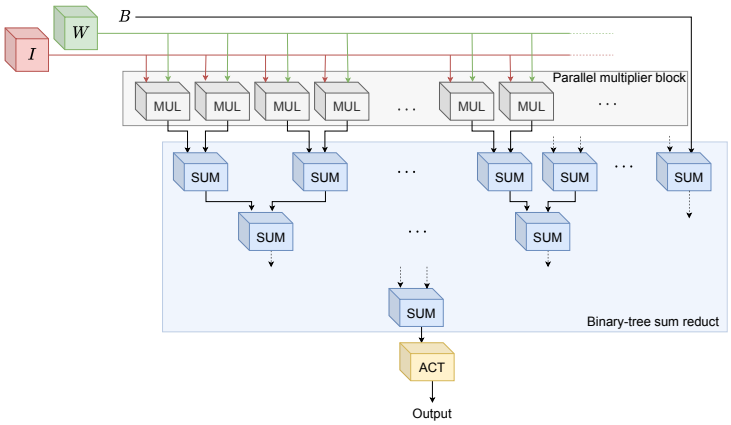



Figure 6.5: RTL block schema of PEs

data-width for inputs, weights, biases, and output; (ii) the input-volume size of the neuron, in terms of depth, height, and width; (iii) the activation-function to be used; (iv) the NABs for multiplications between inputs and weights, and the NABs for partial products sum, in case the precision-scaling technique is adopted; (v) the multiplier and adder to be picked from the EvoApproxLib-Lite [129], in case the inexact-circuit technique is adopted. Flexibility originating from such configuration parameters yield our accelerator generic enough to adapt to several DNN models and approximation degrees. As for the former, the structural configuration of the grid – i.e., how many rows and columns it has – is strictly constrained by the model (and hardware resource). As for the latter, they can be set according to Pareto-optimal configurations resulting from DSE.

#### 6.1.6.4 Case-study #1: validating the method

In order to evaluate our method, we consider the LeNet5 [105] network. LeNet5 [105] is a CNN that perform well in large-scale image processing. It is quite simple CNN when compared to the state-of-the-art architectures: it consists of only five layers (without taking pooling into account) whose characteristics are summarized in Table 6.1, and it requires only 60 thousands parameters to be learned. This makes it the ideal field for several preliminary experiments.

The network being considered has been trained to classify images from the MNIST test data set [107], which consists of a training set of 60000 examples, and a test set of 10000 examples of handwritten digits. The network has been trained using 64 bits floating-point, and exhibits a 99.07% accuracy when classifying images from the mentioned data set. We performed a 8-bit quantization, witnessing no accuracy loss.

Table 6.1: Structural characteristics of layers of LeNet5 [105]

| Layer Layer Type | Input volume | Kernel size | Filters | Activation | Output volume | |
|---|---|---|---|---|---|---|
| Conv1 | Convolution | 1x32x32 | 1x5x5 | 6 | tanh | 6x28x28 |
| Conv2 | Convolution | 6x14x14 | 6x5x5 | 16 | tanh | 16x10x10 |
| Conv3 | Convolution | 16x5x5 | 16x5x5 | 120 | tanh | 120x1x1 |
| Full1 | Fully Connected | 120x1x1 | - | - | tanh | 84x1x1 |
| Full2 | Fully Connected | 84x1x1 | - | - | tanh | 10x1x1 |

In this case-study we configured the Clang-Chimera tool to supersede exact multiplications within the three CLs and two FCLs of LeNet5 using approximate multipliers designed while resorting to the precision-scaling technique. The latter allow selectively introducing approximation by means of configurable parameters. As discussed in Chapter 3, such configuration parameters constitute decision variables for MOP-based DSE; therefore, the Bellerophon tool encodes each approximate configuration, i.e., each individual, using a five element long vector, i.e., using a chromosome composed of five genes. Each of the latter governs the NAB for multiplications within a given layer.

Concerning fitness-functions, we resort to simulations performed on the test dataset in order to assess the classification accuracy-loss due to approximation. As far as hardware-requirements are concerned, we have to take into account

Let us consider CLs. In particular, let $d_i \times h_i \times w_i = I_i$ be the input volume

size of neurons belonging to the i-th CLs, where $d_i$, $h_i$ and $w_i$ being the depth, the height and the width of the volume, respectively, as depicted in Figure 6.6. Thus, according to (6.17), each neuron performs $I_i$ partial products between inputs and synaptic weights, and the minimum amount of additions required to sum $I_i$ operands is given by $\sum_{j=0}^{log_2 I_i+1} \frac{I_i}{2^{j+1}}$. We estimate savings from the ratio of neglected bits over the

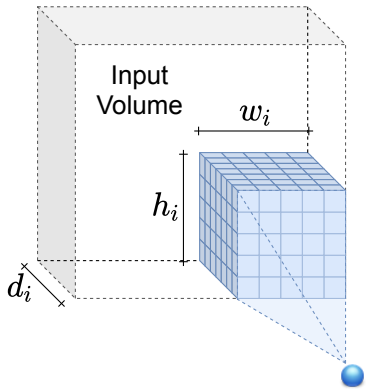



Figure 6.6: Receptive-field of a neuron and its input volume.

exact representation, as given by (6.21), where $S$ be the data-width for inputs, weights, biases, and outputs adopted for the non-approximate network.

$$\frac{2 \times NAB_{mul} \times I_i + \sum_{j=0}^{log_2 I_i+1} \frac{I_i}{2^{j+1}} \times NAB_{add}}{2 \times S \times I_i + S \times \sum_{j=0}^{log_2 I_i+1} \frac{I_i}{2^{j+1}}} \tag{6.21}$$

Please kindly note that the same reasoning still apply to neurons belonging to FCLs, since the latter can be considered as CLs having a mono-dimensional receptive-field. When considering the whole network, savings for the i-th layer can be computed by multiplying (6.21) for the number of neurons within the layer, i.e., for the output volume size $O_i = D_i \times H_i \times W_i$. Let $N$ be the amount of approximate layers, our proposed *reward fitness-function* (6.22) is straightly derived from (6.21), taking into account input and output volumes, and the introduced approximation degree for each approximate layer.

$$\rho = \frac{\sum_{i=0}^{N} \left[ \left( 2 \times NAB_{mul_i} \times I_i + \sum_{j=0}^{log_2 I_i+1} \frac{I_i}{2^{j+1}} \times NAB_{add_i} \right) \times O_i \right]}{S \times \sum_{i=0}^{N} \left[ \left( 2 \times I_i + \sum_{j=0}^{log_2 I_i+1} \frac{I_i}{2^{j+1}} \right) \times O_i \right]} \tag{6.22}$$

Please kindly note that properties of the multiplication operation can be leveraged to slightly simplify Equation (6.22). Indeed, approximating even by a single operand setting to zero the least significant $NAB$ bits, at least $2 \times NAB$ bits of the result will be zero. This can be exploited to set $NAB_{add_i} = 2 \times NAB_{mul_i}$, which drastically reduces the design-space to be searched during DSE. Thus, Equation (6.22) gets simplified as follows.

$$\rho = \frac{\sum_{i=0}^{N} \left[ \left( 2 \times NAB_{mul_i} \times I_i + \sum_{j=0}^{log_2 I_i + 1} \frac{I_i}{2^j} \times NAB_{mul_i} \right) \times O_i \right]}{S \times \sum_{i=0}^{N} \left[ \left( 2 \times I_i + \sum_{j=0}^{log_2 I_i + 1} \frac{I_i}{2^{j+1}} \right) \times O_i \right]} \tag{6.23}$$

To find a suitable NSGA-II configuration, we conducted several DSE campaigns with different parameters. As a result, we deduced two things: (i) to obtain a populous frontier and avoid local sub-optimum, we need to increase the initial population size as much as possible, yet without penalizing too much the time required for exploration; (ii) in order to avoid long-run exploration around local sub-optimum, mutations have to take place frequently; (iii) the number of generations on the one hand must be high enough to ensure that the results of the DSE are close to the optimum, but on the other hand must not be penalizing for the computational time. Hence, we set our GA parameters as follows: initial population equals to 300 individuals, mutation and crossover probabilities both set 0.9, and 13 generation epochs. Finally, we set a maximum error threshold equals to 1% accuracy loss.

Table 6.2 report approximate configurations resulting from the DSE: each row of the tables report one approximate configuration: besides the corresponding error and reward fitness values, for each of the layer we approximate the amount of neglected bits. We can observe that at once the classification accuracy loss is undoubtedly negligible and the estimated reward is quite substantial, thanks to the large inner error resiliency of the considered network. Although too small to be considered significant, in some cases we even obtained an increased accuracy (negative loss).

After the DSE, to correctly evaluate the final gains, we performed FPGA synthesis of Pareto-optimal configurations while targeting a Xilinx Virtex Ultrascale+. These syntheses involve only one single neuron, in order to provide a fair estimation of

hardware requirements, viz. as independent as possible from configuration parameters governing the structure of the accelerator. Furthermore, for the same reason, we disabled advanced FPGA features, e.g. DSPs, during syntheses.

Figure 6.7 report synthesis results: these stacked-bars graphs show, for each of the layers, the amount of FPGA LUTs required by a single neuron. The left-most column refers to the non-approximate configuration, while columns on the right refer to Pareto-optimal approximate configurations from Table 6.2. A significant reduction, actually up to 45%, of required resources in terms of FPGA LUTs can be observed.

As foreseeable, savings achieved due to precision-scaling do not only concern hardware requirements, but also energy consumption. Trivially, the less hardware a circuit require, the less energy is spent to power it, and since the least significant bits of inputs, weights, and biases are always set to zero, the whole approximate circuit is also expected to exhibit a lower switching activity w.r.t. its exact counterpart.

In order to evaluate potential power savings, we performed simulations on the exact neuron and on approximate ones lying on the Pareto-front resulting from the DSE. Simulations involve 10000 input combinations, each consisting of an appropriate amount of inputs, weights and bias vectors, depending on the input volume size of the considered neuron. Figure 6.8 reports simulation results. Again, up to 35% savings in terms of power consumption can be observed.

Table 6.2: DSE results for 8-bits LeNet5

| Conf# | Error (%) | Reward (%) | Conv.1 | Conv.2 | Conv.3 | F.C.1 | F.C.2 |
|---|---|---|---|---|---|---|---|
| 1 | 0.18 | 32.91 | 2 | 3 | 0 | 3 | 2 |
| 2 | 0.34 | 36.25 | 3 | 3 | 2 | 0 | 2 |
| 3 | 0.48 | 37.31 | 3 | 3 | 2 | 3 | 4 |
| 4 | -0.06 | 24.60 | 2 | 2 | 0 | 2 | 3 |
| 5 | 0.07 | 28.61 | 3 | 2 | 2 | 1 | 3 |
| 6 | 0.35 | 36.28 | 3 | 3 | 2 | 0 | 3 |
| 7 | 0.10 | 28.86 | 3 | 2 | 2 | 2 | 0 |
| 8 | 0.38 | 36.99 | 3 | 3 | 1 | 3 | 0 |
| 9 | 0.12 | 32.60 | 2 | 3 | 0 | 2 | 3 |
| 10 | -0.02 | 24.94 | 2 | 2 | 2 | 2 | 0 |
| 11 | -0.07 | 16.64 | 0 | 2 | 1 | 1 | 3 |

Energy consumption due to memory operations is also affected due to precision-scaling. Indeed, the lower the data width for inputs, weights and biases, the lower energy required to move data back and forth from the memory to the accelerator.

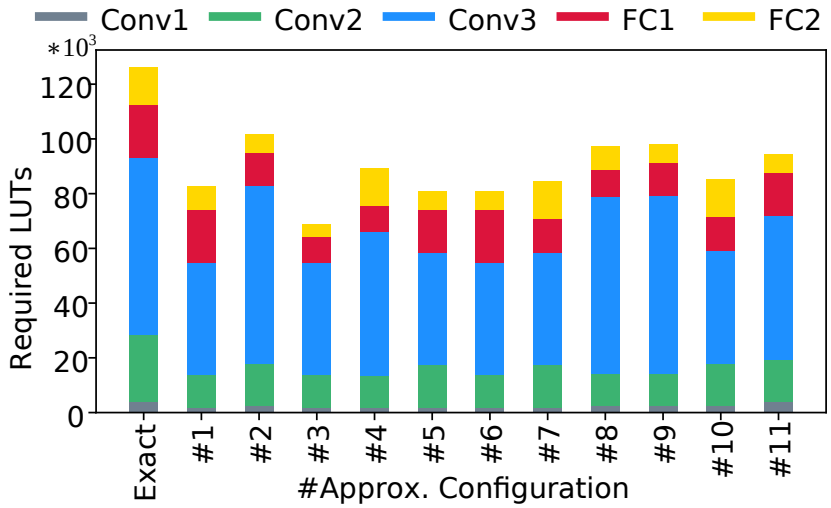



Figure 6.7: Required LUTs for 8-bits LeNet5 while targeting a Xilinx Virtex Ultrascale+ FPGA

Let suppose moving a bit from the computer system memory to the hardware accelerator will cost 1 Energy Unit (EU). Thus, moving inputs, weights and biases from the main memory to a hardware accelerator implementing a neuron for a CL layer in LeNet5 would require $2 \times S \times I + S$ EUs, where $S$ is the data width and $I$ the input volume size. Moving the neuron's output from the accelerator to memory would require $S$ EUs. On the other hand, had the precision scaling reduced data width of $NAB_{mul}$ and $NAB_{add}$, the same operations would require $2 \times (S - NAB_{mul}) \times I + (S - NAB_{add})$ EUs and $S - NAB_{add}$ EUs, respectively. Thus, savings due to memory operations can be computed as reported in Equation (6.24).

$$\rho_m = \frac{2 \times \sum_i^N [(S - NAB_{mul_i}) \times I_i] + \sum_i^N (S - NAB_{add_i})}{2 \times S \times \sum_i^N (I_i + 1)} \tag{6.24}$$

Table 6.3 reports the estimation of potential energy savings due to memory reads and write operations, estimated using Equation (6.24). For the reader convenience, the same Table also summarizes savings due to the reduced hardware requirements. These preliminary results suggest the MOP-based DSE is able to efficiently evaluate configurations within the design space, and provide the designed with suitable trade-offs for the design of actual accelerators.

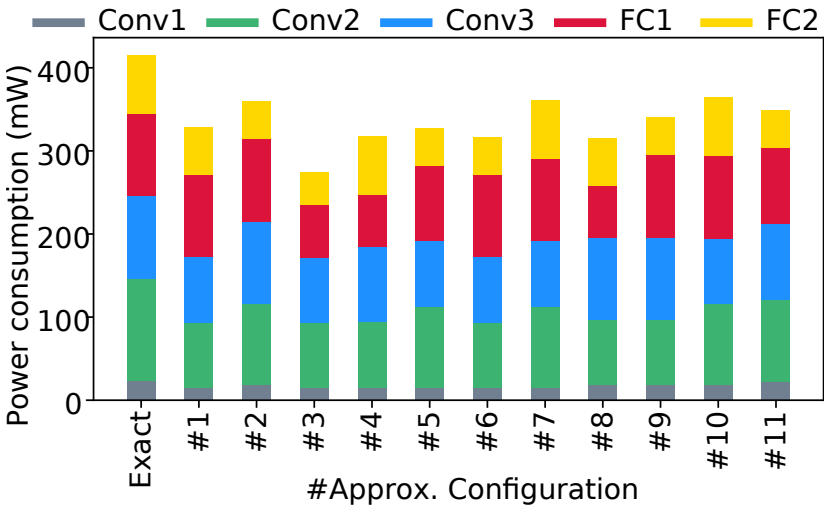



Figure 6.8: Estimated power consumption for 8-bits LeNet5 while targeting a Xilinx Virtex Ultrascale+ FPGA

#### 6.1.6.5 Case-study #2: approximating sums with degree independent of multiplications

As we discussed in the previous Section, properties of the multiplication operation can be leveraged to slightly simplify Equation (6.22), i.e., it is possible to impose $NAB_{add_i} = 2 \times NAB_{mul_i}$ obtaining Equation (6.23). This drastically reduces the solution space to be searched during DSE.

Almost spontaneously, the question arises whether the solutions that would be obtained by setting $NAB_{add_i}$ independently w.r.t. $NAB_{mul_i}$ can be better than those obtained by setting $NAB_{add_i} = 2 \times NAB_{mul_i}$. Thus, we configured Clang-Chimera to truncate input operands of both multiplications and additions within CLs and FCLs, and we configured the Bellerophon to vary $NAB_{add_i}$ independently w.r.t. $NAB_{mul_i}$ within each of the layers, while using Equation (6.22) rather than Equation (6.23), to estimate savings. Finally, we configured the DSE using the same NSGA-II parameters we adopted in the previous case-study, for a fair comparison.

Table 6.4 report results from DSE: besides the corresponding error and reward fitness values, each row reports the amount of neglected bits for additions and multipli-

Table 6.3: Summary of savings for 8-bits LeNet5

| Conf | FPGA LUTs | FPGA Power Consumption (mW) | Memory Power Consumption (EU) | FPGA LUTs Savings (%) | FPGA Power Savings (%) | Memory Power Savings (%) |
|---|---|---|---|---|---|---|
| 0 | 126185 | 415.48 | 12504 | - | - | - |
| 1 | 82895 | 327.81 | 10438 | 34.31 | 21.10 | 16.52 |
| 2 | 101777 | 359.69 | 9508 | 19.34 | 13.43 | 23.45 |
| 3 | 68770 | 273.80 | 8447 | 45.50 | 34.10 | 32.45 |
| 4 | 89173 | 318.11 | 10811 | 29.33 | 23.44 | 13.54 |
| 5 | 80890 | 327.87 | 9399 | 35.90 | 21.09 | 24.83 |
| 6 | 80894 | 316.42 | 9339 | 35.89 | 23.84 | 25.31 |
| 7 | 84458 | 361.14 | 9665 | 33.07 | 13.08 | 22.70 |
| 8 | 97252 | 315.24 | 9924 | 22.93 | 24.13 | 20.63 |
| 9 | 98059 | 340.35 | 10510 | 22.29 | 18.08 | 15.95 |
| 10 | 85064 | 364.39 | 9716 | 32.59 | 12.30 | 22.30 |
| 11 | 94211 | 348.40 | 10535 | 25.34 | 16.14 | 17.20 |

cations for each of the layers, respectively in the $\sum$ and $\times$ columns.

At the end of DSE, we performed FPGA synthesis of Pareto-optimal configurations from Table 6.4 in order to correctly evaluate the final gains. Again, these syntheses involve only one single neuron. We also performed simulations on the exact neuron and on the approximate neuron configurations lying on the Pareto-front resulting from the DSE in order to estimate power consumption. As before, simulations involve 10000 input combinations, each consisting of an appropriate amount of inputs, weights and bias vectors depending on the input volume size of the considered neuron. Figure 6.9 and 6.10 report results in terms of required FPGA LUTs and power consumption, respectively.

Table 6.4: DSE results for the 8-bits of LeNet5 while approximating both additions and multiplications

| | | | Neglected Bits | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Conv. 1 | | Conv. 2 | | Conv. 3 | | F.C.1 | | F.C. 2 | |
| Conf # | Error (%) | Reward (%) | $\sum$ | $\times$ | $\sum$ | $\times$ | $\sum$ | $\times$ | $\sum$ | $\times$ | $\sum$ | $\times$ |
| 1 | -0.09 | 18 | 0 | 0 | 3 | 2 | 2 | 1 | 0 | 0 | 1 | 4 |
| 2 | -0.08 | 30 | 2 | 1 | 3 | 2 | 3 | 2 | 4 | 2 | 0 | 4 |
| 3 | 0.02 | 31 | 2 | 2 | 2 | 2 | 4 | 2 | 6 | 1 | 1 | 4 |
| 4 | 0.17 | 32 | 6 | 3 | 6 | 0 | 4 | 2 | 0 | 3 | 6 | 3 |
| 5 | 0.19 | 33 | 1 | 0 | 5 | 3 | 4 | 2 | 5 | 1 | 1 | 3 |
| 6 | 0.22 | 35 | 2 | 2 | 3 | 3 | 4 | 2 | 6 | 1 | 2 | 4 |
| 7 | 0.30 | 36 | 3 | 1 | 4 | 0 | 6 | 2 | 1 | 4 | 5 | 2 |
| 8 | 0.48 | 40 | 5 | 3 | 6 | 3 | 4 | 2 | 5 | 3 | 3 | 4 |

In order to state which Pareto-front provides better results, we resort to the *Coverage of two sets* metric, proposed in [189]. Let *A* and *B* be two sets of non-dominated solutions for a MOP. The function $\mathcal{C}$ in Equation (6.25) maps the pair $(A,\ B)$ to the interval $[0, 1]$: where the expression $\alpha$ *covers* $\beta$ ($\alpha \succeq \beta$) means that the solution $\alpha$ *dominates* the solution $\beta$ or they are *the same* solution. The value $\mathcal{C}(A, B) = 1$ means

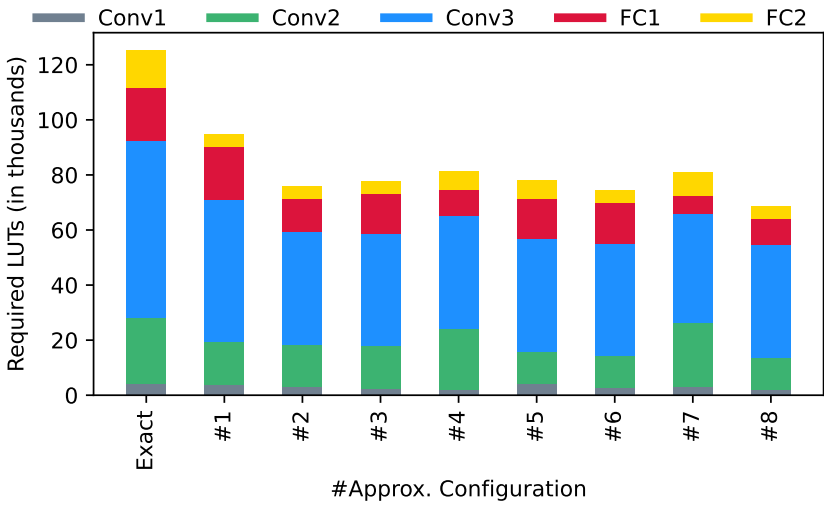



Figure 6.9: Required LUTs for 8-bits LeNet5 while approximating both additions and multiplications

that all points in $B$ are dominated by or equal to points in $A$. Conversely, the value $\mathcal{C}(A,B) = 0$ represents the situation where no points in $B$ are covered by any points in $A$. When using this metric, both $\mathcal{C}(A,B)$ and $\mathcal{C}(B,A)$ have to be considered, as they are not necessarily equal.

$$\mathcal{C}(A,B) := \frac{|\{\forall \beta \in B;\ \exists\, \alpha \in A : \alpha \succeq \beta\}|}{|B|} \tag{6.25}$$

We measured $\mathcal{C}$ between configurations from Table 6.2 and Table 6.4 while using classification-accuracy loss and actual hardware requirements – i.e., the actual LUTs and power consumption resulting from syntheses and simulations – to define the two Pareto-fronts. As reported in Table 6.5, we obtained 0 and 22% coverage, which means that varying $NAB_{add_i}$ independent w.r.t. $NAB_{mul_i}$ provides configurations which never dominate those provided by setting $NAB_{add_i} = 2 \times NAB_{mul_i}$. Vice-versa, approximate configurations obtained by setting $NAB_{add_i} = 2 \times NAB_{mul_i}$ dominates those provided by varying $NAB_{add_i}$ independent w.r.t. $NAB_{mul_i}$ 22% of time. Thus, we can conclude that, for the same amount of effort spent on the DSE, configuring the degree of approximation for the sums by exploiting the properties of multiplication – i.e., setting $NAB_{add_i} = 2 \times NAB_{mul_i}$, – yields better solutions. This is due to the

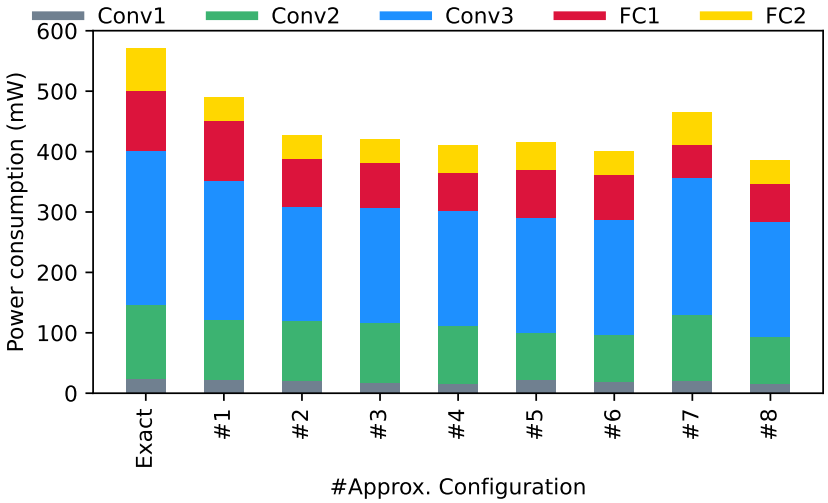



Figure 6.10: Estimated power consumption for 8-bits LeNet5 while approximating both additions and multiplications

increased size of the design space when $NAB_{add_i}$ does not depend on $NAB_{mul_i}$.

Table 6.5: *Coverage of two sets* metric between Pareto-fronts in Table 6.2 and Table 6.4

| **Pareto-fronts** | $\mathcal{C}$ (A, B) |
|---|---|
| add & mul v.s. mul only | 0 |
| mul only v.s.add & mul | 0.22 |

#### 6.1.6.6 Case-study #3: investigating the relationships between data-width and error-resilience.

From a mathematical standpoint, the quantization can be seen as a mapping between $A \subset \mathbb{R} \rightarrow B \subset \mathbb{Z}$, which implies $|A| > |B|$, where $|\cdot|$ is the cardinality of a set, i.e., its size in terms of elements it contains. As it is easy to guess, reducing the space on which weights (and inputs, of course) are represented could negatively affect the error-resiliency of the network, hence its capability to be approximated. Thus, the less the size of $B$, we expect the less the network can be approximate.

Though, in this experiment we compare the resilience to error and the consequent savings that can be obtained from a different implementation of LeNet5, quantized to

16 bits, rather than of 8 bits.

As done in Section 6.1.6.4, we configure Clang-Chimera to truncate multiplications in the three CLs and in the two FCLs of a 16-bits quantized LeNet5 implementation. Thus, the tool generates an approximate version of the considered CNN in which it is possible to configure, for each multiplication involved in the weighted sum, the NABs, in order to tune the introduced approximation degree. In order to estimate the error due to the approximation, we configured Bellerophon to execute the approximate CNN on the test data, in order to assess the classification-accuracy loss. Concerning the *reward* fitness-function, we estimate the gains by using Equation (6.23). Furthermore, in order to avoid biasing, we configured the DSE using the same NSGA-II parameters we adopted in Section 6.1.6.4.

Table 6.6 report approximate configurations resulting from the DSE: each row of the tables report one approximate configuration, i.e., the amount of neglected bits, along with the corresponding error and reward fitness values. Then, we performed FPGA synthesis and simulations in order to estimate hardware requirements both in terms of LUTs and power consumption, which are reported in Figure 6.11 and Figure 6.12. As the reader can observe, up to 75% and 55% savings in terms of LUTs and power consumption, respectively, can be achieved, suggesting the error resiliency, hence the ability to be approximated, strongly depends on the data size.

Table 6.6: DSE results for the 16-bits implementation of LeNet5

| Conf# | Error (%) | Reward (%) | Conv.1 | Conv.2 | Conv.3 | F.C.1 | F.C.2 |
|---|---|---|---|---|---|---|---|
| 1 | -0.06 | 44 | 5 | 9 | 6 | 9 | 11 |
| 2 | -0.05 | 51 | 2 | 6 | 9 | 5 | 12 |
| 3 | -0.04 | 54 | 2 | 6 | 0 | 6 | 12 |
| 4 | 0.11 | 55 | 9 | 10 | 10 | 6 | 4 |
| 5 | 0.12 | 57 | 9 | 10 | 10 | 7 | 5 |
| 6 | 0.13 | 60 | 1 | 10 | 10 | 5 | 9 |

At this point, the question arises whether to approximate a network quantized on more bits, e.g., 16-bits, can originate approximate configurations that require less hardware resources than those originating from a network quantized on fewer bits, e.g., 8-bits. To answer this question, we compare the Pareto-fronts from Table 6.2 and Table 6.6, in order to state which implementation provides better Pareto-front. Again, we resort to the *Coverage of two sets* metric [189], reported in Equation (6.25). We measured $\mathcal{C}$ between configurations from Table 6.2 and Table 6.6 while using classification-accuracy loss and actual hardware requirements – i.e., the actual LUTs

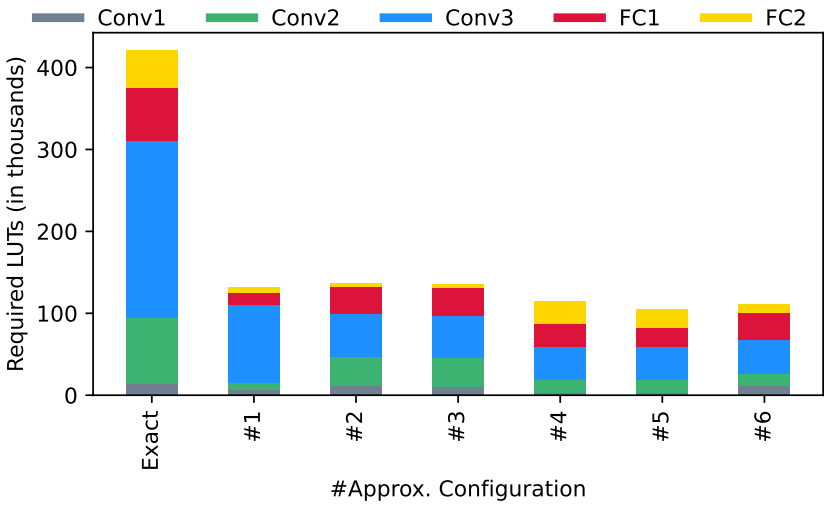



Figure 6.11: Required LUTs for 16-bits LeNet5 while targeting a Xilinx Virtex Ultrascale+ FPGA

and power consumption resulting from syntheses and simulations – to define the two Pareto-fronts. Table 6.7 reports results of such a comparison: we obtained 1 and 0 coverage, which means solutions from Table 6.2 – i.e., those resulting from the 8-bits implementation of LeNet5 – always dominates those from Table 6.6 – i.e., those resulting from the 16-bits implementation of LeNet5 – while the vice-versa never occurs. This suggests that employing implementations quantized to fewer bits results in lower hardware requirements, albeit implementations using quantization to more bits brings much more room for approximation.

Table 6.7: *Coverage of two sets* metric between Pareto-fronts in Table 6.2 and Table 6.6

| **Pareto-fronts** | $\mathcal{C}(\mathrm{A}, \mathrm{B})$ |
|---|---|
| 8b vs. 16b. | 1 |
| 16b vs. 8b | 0 |

#### 6.1.6.7 Impact of precision-scaling on hardware components

At this point it is natural to wonder what are the reasons for which you can get similar savings both in terms of area on silicon and in terms of energy consumption. In this

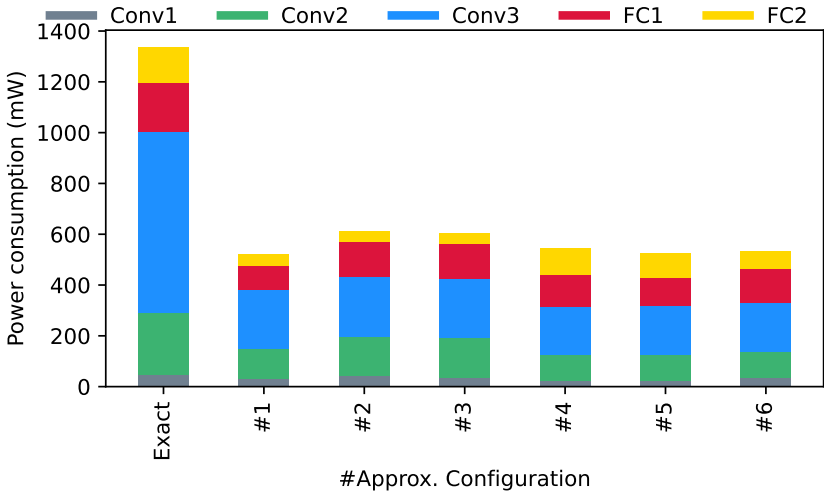



Figure 6.12: Estimated power consumption for 16-bits LeNet5 while targeting a Xilinx Virtex Ultrascale+ FPGA

Section we will try to give an explanation to these phenomena, looking at the effects of the precision-scaling technique on the accelerator hardware implementation that we discussed earlier.

In order to make PEs easily adaptable to any data-width and any approximation degree, we chose two of the most commonly adopted hardware implementations for multipliers and adders, i.e., the MAC-cell multiplier and the ripple-carry adder. In order to understand the impact of precision scaling on such components, let us consider a ripple-carry adder first. Suppose the three least significant bits of both operands are set to zero, i.e., $NAB_{add} = 3$. So, the output of full-adder cells computing the three least significant bits of the sum is always zero and, therefore, they can be superseded by constant-zero, thus reducing hardware costs, as depicted in Figure 6.13. Savings, in this case, are proportional to $NAB_{add}$, i.e., $\rho \propto NAB_{add}$. Consider, now, the MAC-

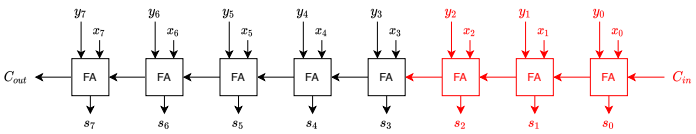



Figure 6.13: Effect of precision-scaling on a ripple-carry adder. Constant-zero cells are highlighted in red.

multiplier, and, again, let us suppose the three least significant bits of both operands are set to zero, i.e., $NAB_{mul} = 3$: cells producing a value which is always equal to zero are highlighted in red in Figure 6.14. As we can observe, the MAC-cells on the first $NAB_{mul}$ rows produce always values equal to zero, so they can be superseded by a constant-zero signal, reducing hardware costs proportionally to $S \times NAB_{mul}$, where $S$ is the data width. Parts of subsequent rows are also affected by approximation, providing further savings, proportional to the following.

$$NAB_{mul}+NAB_{mul}-1+NAB_{mul}-2+\cdots+1 = \sum_{i=1}^{NAB_{mul}} i = \frac{NAB_{mul}^2 + NAB_{mul}}{2} \tag{6.26}$$

This means $\rho \propto NAB_{mul}^2$. Therefore, area requirements of a 16 bits multiplier with $NAB_{mul} = 8$, for instance, are not halved, indeed the actual size of the multiplier is one third of that required by its exact counterpart, which significantly boosts savings of the whole neuron. Nevertheless, the quadratic term is never prominent over the linear one when 8-bits quantization is performed, since the former is always less than the latter in the allowed (0, 8) variation range for $NAB_{mul}$.

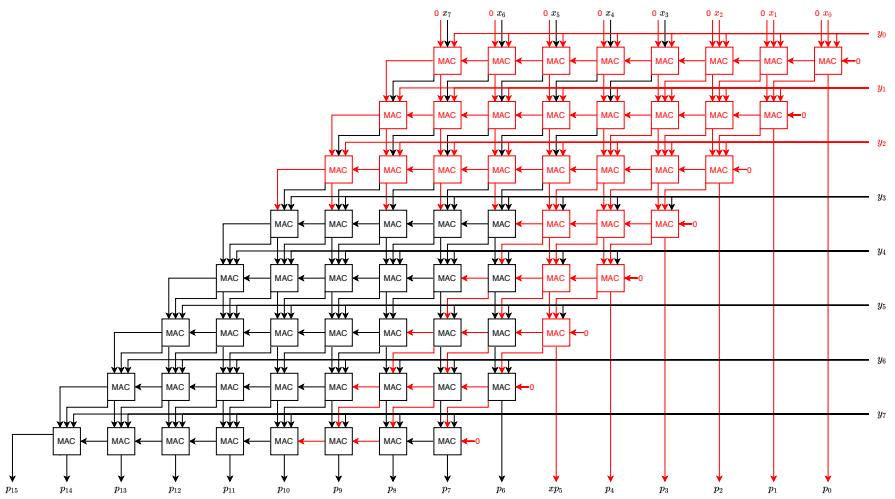



Figure 6.14: Effect of precision-scaling on a MAC multiplier. Constant-zero cells are highlighted in red.

#### 6.1.6.8 Case-study #4: building complex architectures from elementary approximate components

In this case-study we configure the Clang-Chimera is configured to supersede exact multiplications in the three convolutional and in the two fully connected layers using approximate circuits taken from the EvoApproxLib-Lite [129]. Thus, the tool generates an approximate version of the considered CNN in which it is possible to select, for each multiplication involved in the weighted sum computation within neurons, an implementation between either the exact or an approximate implementation from the mentioned library.

Concerning MOP, as done for precision-scaling, one of the objectives of the Bellerophon is to find approximate solutions minimizing the classification-accuracy loss. Therefore, in order to perform error assessment, we configured Bellerophon to execute the approximate CNN to obtain its classification accuracy on the MNIST test data set [107]. Then, that accuracy is compared against the accuracy of the non-approximate 16-bits quantized CNN, and the error is computed as the difference between the two CNNs accuracy.

We estimate the circuit area by taking into account (i) the silicon area of exact and approximate multipliers, as reported in [129], (ii) the input-volume of a neuron, which impacts the amount of operations performed within it, and (iii) the amount of neurons within a layer, i.e. the output volume size of a layer. In details, being: (i) $N$: the amount of approximate layers, (ii) $I_i = d_i \times h_i \times w_i$: the input volume size of each neuron belonging to the i-th layer, (iii) $O_i = D_i \times H_i \times W_i$: the i-th layer output volume size, (iv) $\alpha_i$: the silicon area of the multiplier being adopted within i-th layer as reported in [129], the Bellerophon aims at minimizing Equation (6.27), which is the sum of the silicon area of multipliers being adopted within each of the layers, weighted according to input and output volume size of each layer.

$$\rho = \sum_{i}^{N} \alpha_i \times I_i \times O_i \tag{6.27}$$

Furthermore, we also aim to minimize the power consumption of the circuit, which is estimated resorting to the same reasoning as for silicon-area minimization. In particular, being $\beta_i$: the power consumption of the multiplier adopted within i-th layer,

the Bellerophon aims at minimizing Equation (6.28), which is the sum of the power consumption of multipliers being adopted within each layer, weighted according to the input and output volume size.

$$\psi = \sum_{i}^{N} \beta_i \times I_i \times O_i \tag{6.28}$$

Resulting approximate configurations are reported in Table 6.8. For each of the configurations, besides the corresponding fitness-function values, also the multiplier circuits being adopted are reported. These approximate configurations are, then, employed to configure the above-mentioned accelerator; then, we performed ASIC synthesis targeting the 65$\mu m$ FinFET library to measure the actual hardware requirements entailed by the approximation. Figure 6.15 and Figure 6.16 report silicon-die area requirements

Table 6.8: Bellerophon results for LeNet5 while using approximate circuits from [129]

| **Conf#** | **Error (%)** | **Silicon area** ($\mu m^2$) | **Power (W)** | **Conv.1** | **Conv.2** | **Conv.3** | **F.C.1** | **F.C.2** |
|---|---|---|---|---|---|---|---|---|
| 1 | -0.04 | $1012.24 \times 10^6$ | 945.199 | Exact | HHP | Exact | GK2 | HDG |
| 2 | 0.15 | $900.95 \times 10^6$ | 736.031 | HFZ | HDG | Exact | HDG | Exact |
| 3 | -0.08 | $1143.84 \times 10^6$ | 1014.180 | Exact | GK2 | G7Z | HHP | HFZ |
| 4 | 0.13 | $950.48 \times 10^6$ | 789.560 | HFZ | G7Z | G80 | Exact | G7Z |
| 5 | 0.04 | $970.83 \times 10^6$ | 859.568 | HDG | HEB | HHP | G7F | G7Z |
| 6 | 0.07 | $970.17 \times 10^6$ | 784.903 | G80 | G7F | G80 | HEB | G7F |
| 7 | 0.01 | $997.27 \times 10^6$ | 832.993 | G80 | HDG | Exact | GK2 | HFZ |
| 8 | -0.02 | $999.64 \times 10^6$ | 930.959 | Exact | HHP | HHP | GK2 | HDG |
| 9 | 0.22 | $814.35 \times 10^6$ | 648.202 | HDG | HFZ | Exact | HFZ | HFZ |
| 10 | 0.34 | $731.97 \times 10^6$ | 563.760 | HFZ | HFZ | GK2 | HHP | G80 |

and estimated power consumption provided by the Cadence Genus Synthesis Solution tool. As discussed above, the third convolutional layer (Conv.3) is the most burdensome, since its input volume size. Anyway, the predicted savings are still confirmed: using our methodology results in hardware resources significantly decrease as the introduced classification error increases. Configuration #8, for instance, allows achieving up to 35% and 30% savings in terms of silicon area and power consumption.

**6.1.6.8.1 Comparison with previous works** Authors of [129] proposed a comparable approach in which basic arithmetic components are first approximate while considering component-based error and saving metrics, thus employed to design a

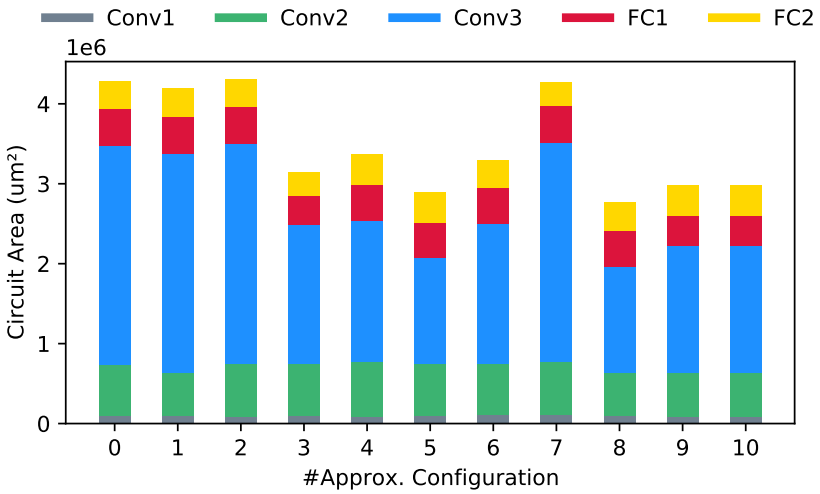



Figure 6.15: Silicon-die area for 16-bits LeNet5 while using multipliers from the EvoApproxLib-Lite library [129]

hardware accelerator CNNs. In particular, multipliers are employed to perform error-resiliency estimation of single layers of CNNs, and to reduce figure of merits such as silicon area and power consumption of hardware accelerators.

Although the results are not directly comparable, those achieved through the use of our methodology – 0% classification accuracy drop against 35% area and 30% power savings – are on the same quantitative relation as those obtained using state-of-the-art approaches – 1.8% accuracy traded for 29% power savings. On the other hand, looking at the methodological aspect, we deem that our approach provides a significant step forward. In fact, it allows considering the application as a whole, and to explore, in a single run, the different degrees of approximation that each of the layers of the network is able to withstand, allowing to diversify the degree of approximation that can be exploited during the design of a hardware accelerator. Conversely, the state-of-the-art studies carry out the analysis of resilience with respect to the error by considering the layers one at a time, and, as far as the design of an accelerator is concerned, they do not fully exploit the approximation that is possible to introduce, employing the same degree of approximation for all the layers. Moreover, they do not allow exploring the effects of different approximation techniques on the same application.

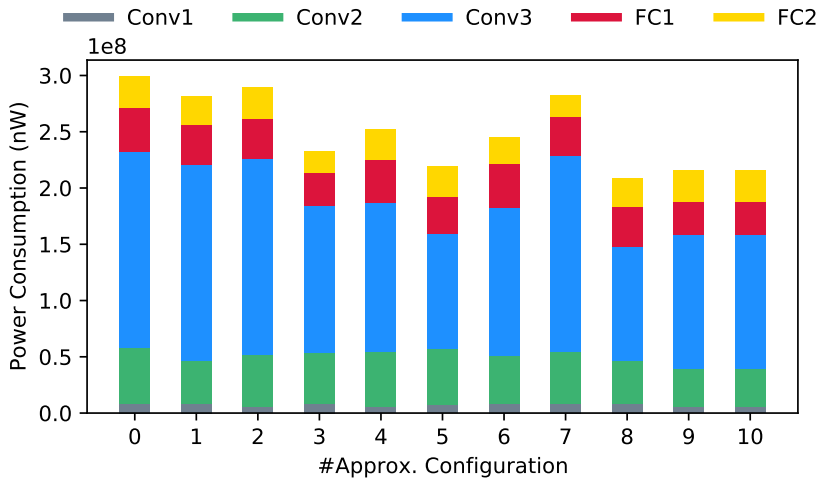



Figure 6.16: Power consumption for 16-bits LeNet5 while using multipliers from the EvoApproxLib-Lite library [129]

**6.1.6.8.2 Comparing precision-scaling and approximate-circuits techniques** In this section, we make a comparison between the precision-scaling and the approximate circuit techniques, while resorting to the case-study discussed above.

Resorting to the 16-bits quantized CNN model we adopted for the approximate circuit technique case study, here we configure Clang-Chimera to truncate input operands and results of multiplications in the three convolutional and in the two fully connected layers of the considered network. As a result, the tool generates an approximate version of the considered CNN in which it is possible to configure, for each multiplication involved in the weighted sum, the number of neglected bits, in order to tune the introduced approximation degree. Concerning the DSE phase, we configured Bellerophon to minimize both the classification-accuracy loss and silicon-die area. In particular, (i) Bellerophon compares the approximate and the non-approximate networks in terms of classification accuracy while considering the MNIST test data set, in order to assess error, and (ii) for what pertains to area minimization, we resort to Equation (6.23). Finally, we synthesized approximate configurations resulting from Bellerophon while targeting the 65 $\mu m$ FinFET technology library – the same we targeted for the approximate circuit case study – in order to compare actual savings provided by the adopted approximate techniques, both in terms of silicon area and power consumption.

Results of such a comparison are reported in the classification-accuracy loss vs silicon area and power consumption perspectives, respectively in Figure 6.17 and Figure 6.18. In order to state which Pareto-front provides better results, we again resort to the *Coverage of two sets* metric [189], reported in Equation (6.25). We measured $\mathcal{C}$ between the Pareto-fronts resulting from precision-scaling and approximate circuits, and vice-versa. As reported in Table 6.9, we obtained 90% and 22% coverage, which means the precision-scaling dominates the approximate circuits techniques 90% of time, while the opposite occurs only 22% of time. Although the precision-scaling seems to

Table 6.9: *Coverage of two sets* metric between Pareto-fronts

| **Pareto-fronts** | $\mathcal{C}(\mathrm{A}, \mathrm{B})$ |
|---|---|
| Ax Circuits v.s. Precision scaling | 0.22 |
| Precision scaling v.s. Ax Circuits | 0.91 |

provide better trade-offs w.r.t the approximate-circuits technique, results might differ whether considering a different application domain, paving the way for future research.

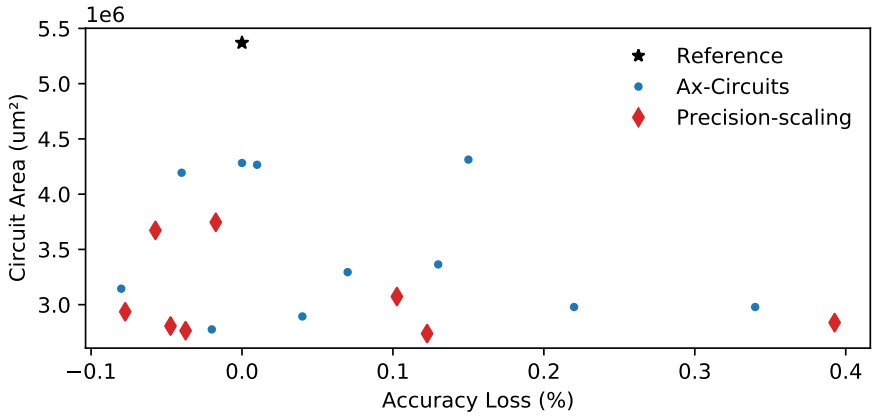



Figure 6.17: Comparison between precision-scaling and approximate circuits techniques: error v.s. silicon area

#### 6.1.6.9 Case-study #5: applying the loop-perforation technique

The opportunity to attempt applying the loop-perforation technique arise from the particular C/C++ implementation of CLs and FCLs provided by the N2D2 framework.

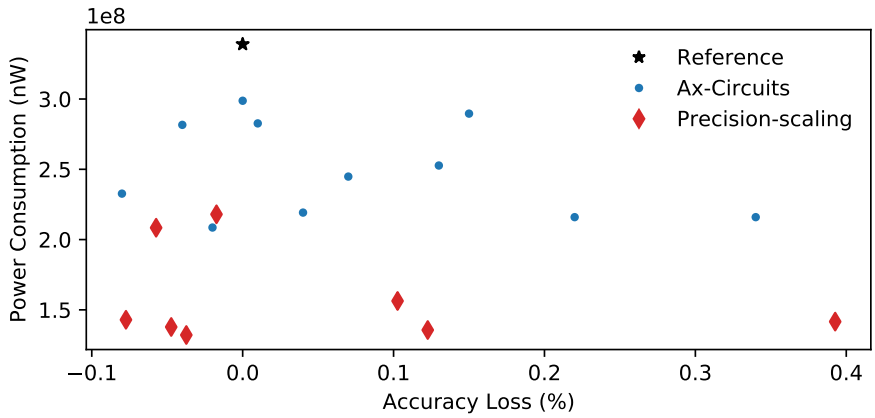



Figure 6.18: Comparison between precision-scaling and approximate circuits techniques: error v.s. power-consumption

Listing 6.1 reports an example code for CLs, albeit the implementation of FCLs follows the same concept. CLs are implemented by making use of six nested *for* loops: the three outermost loops iterate on the three dimensions of the output volume, while the three innermost loops perform the weighted sum of the inputs for a single neuron.

Depending on which loop is the target, the effect has an easily imaginable physical interpretation. When applied to outer loops, the effect is to skip the computation of neurons that are on the same row or column of the same depth-slice, or those that compose the same fiber of the output volume of the layer under consideration. In the other case, i.e., when applied to internal cycles, the technique affects the calculation of the weighted sum of each of the neurons of the layer. Depending on the particular cycle, the elements of t he receptive-field of a neuron that are on the same row, column, or fiber are skipped. Obviously, approximating multiple cycles means summing the effects due to the approximation of each one.

```
1  void conv_layer(...) {
2    for (int output = 0; output < output_depth; ++output)
3      for (int oy = 0; oy < output_height; ++oy)
4        for (int ox = 0; ox < output_width; ++ox) {
5          const int ix = (ox * strideX) - paddingX;
6          const int iy = (oy * strideY) - paddingY;
7          SUM_T ws = bias[output];
8          for (int ch = 0; ch < input_depth; ++ch)
9            for (int sy = 0; sy < kernel_height; ++sy)
10             for (int sx = 0; sx < kernel_width; ++sx)
11               ws += W[output][ch][sy][sx] * I[ch][iy + sy][ix + sx];
12         outputs[output][oy][ox] = activation_func(weightedSum);
13     }
14   }
```

Listing 6.1: Implementation of a CL

The Clang-Chimera tool provides two different mutation-operators implementing the loop-perforation technique, namely *Loop1* and *Loop2*. Both introduce a configuration parameter that serves to tune the degree of introduced approximation, i.e., the amount of skipped iterations. Nevertheless, they are slightly different in the introduced behavior. In facts, the former mutator alters the increment of the loop-counter, which is incremented depending on the value of the configuration parameter. The latter mutator alters the body of the loop so that it is executed only when the current value of the loop counter is a multiple of the configuration parameter. This means the degree of approximation introduced by *Loop1* is directly proportional to the value of the configuration parameter, while the proportionality is inverse for *Loop2*. This difference is crucial during the DSE and must be taken into account in order to properly define the reward fitness-function. Anyway, the configuration parameter can be freely set during the DSE for both mutators.

To carry out experiments on loop-perforation, the Clang-Chimera has been configured to perform mutations on CLs only, generating four approximate configurations while combining loops to approximate – i.e, either outer or inner loops – and mutation operators – i.e, either the *Loop1* or *Loop2* mutator. Albeit the error fitness-function we defined in Section 6.1.6.2 still applies, the reward fitness-function and the MOP-encoding to drive the DSE have to be adapted to the loop-perforation technique. Concerning the "reward" fitness-function, it is defined as the ratio between the number of skipped iterations and the total amount of iterations when applying the *Loop1* mutator. Conversely, when applying the *Loop2* mutator, the product between the total amount

of iterations and the frequency with which the body of a loop is executed is a good estimation for the "reward". As for the MOP encoding, approximate variants can be described using a chromosome consisting of 9 genes – one gene for each of the pierced loops within each of the three approximate CLs – each governing the amount of skipped iterations for the corresponding loop.

Albeit the error fitness-function we defined in Section 6.1.6.2 still applies, the reward fitness-function and the MOP-encoding to drive the DSE have to be adapted to the loop-perforation technique. Concerning the "reward" fitness-function, it is defined as the ratio between the number of skipped iterations and the total amount of iterations when applying the *Loop1* mutator. Conversely, when applying the *Loop2* mutator, the product between the total amount of iterations and the frequency with which the body of a loop is executed is a good estimation for the "reward". As for the MOP encoding, approximate variants can be described using a chromosome consisting of 9 genes – one gene for each of the pierced loops within each of the three approximate CLs – each governing the amount of skipped iterations for the corresponding loop.

Table 6.10 and Table 6.12 report DSE results while the *Loop1* and the *Loop2* mutators on outer loops, respectively. Table 6.11 and Table 6.13, instead, refer to approximate configuration resulting from piercing inner loops *Loop1* and the *Loop2* mutators, respectively. As the reader can observe, albeit the amount of skipped iterations is quite modest, the network loses its ability to correctly recognize and classify the elements.

In further experiments, we apply the loop-perforation technique to a single layer at a time, obtaining twelve approximate variants of the code reported in Listings 6.1. Nevertheless, we omit DSE results since they are quite similar to those already discussed: a negligible reward – i.e., skipped iterations – corresponds to a catastrophically high classification error.

These negative results can be easily understood by bearing in mind the physical interpretation of the effects of the technique we discussed above: applying the loop-perforation on one of the external cycles results in not computing elements of the output-volume along the same dimension (height, width, or depth), as if the neurons that compute elements of the activations map are dead. Likewise, for internal cycles, when computing the weighted sum and the activation function the neurons do not take

into account all the input-volume elements that are along the same dimension, as if the corresponding weight is zero. The effects of all these small, but numerous, losses of information add up and dramatically affect the accuracy of classification.

| Error (%) | Reward (%) | Stride 0 | Stride 1 | Stride 2 | Stride 3 | Stride 4 | Stride 5 | Stride 6 | Stride 7 | Stride 8 |
|---|---|---|---|---|---|---|---|---|---|---|
| 87.72 | 24.5 | 3 | 9 | 0 | 9 | 8 | 9 | 4 | 8 | 6 |
| 79.12 | 8.0 | 1 | 1 | 2 | 8 | 2 | 1 | 3 | 9 | 0 |

Table 6.10: Outer-loops approximation using the Loop1 operator.

| Error (%) | Reward (%) | Stride 0 | Stride 1 | Stride 2 | Stride 3 | Stride 4 | Stride 5 | Stride 6 | Stride 7 | Stride 8 |
|---|---|---|---|---|---|---|---|---|---|---|
| 75.58 | 37.5 | 3 | 5 | 1 | 5 | 4 | 4 | 1 | 1 | 9 |
| 73.03 | 17.1 | 6 | 4 | 3 | 1 | 7 | 0 | 7 | 2 | 3 |

Table 6.11: Inner-loops approximation using the Loop1 operator.

| Error (%) | Reward (%) | Stride 0 | Stride 1 | Stride 2 | Stride 3 | Stride 4 | Stride 5 | Stride 6 | Stride 7 | Stride 8 |
|---|---|---|---|---|---|---|---|---|---|---|
| 87.64 | 30.61 | 4 | 0 | 8 | 1 | 2 | 0 | 8 | 6 | 2 |
| 86.77 | 18.63 | 1 | 1 | 2 | 8 | 2 | 1 | 3 | 9 | 0 |

Table 6.12: Outer-loops approximation using the Loop2 operator.

## 6.2 Decision-Tree based Multiple Classifier Systems

DTs stand out for their simplicity and high interpretability level, placing them as one of the most widely used classifier model [179]. A DT is a white-box classification model representing its decisions through a tree-like structure composed of an internal set of nodes containing *test conditions*, and leaf nodes which represent *class labels* [30]. Nodes are joined by arcs symbolizing possible outcomes of each test condition. Classes can be either categorical or numerical. In the former case, we refer to classification trees, while in the latter case, we refer to regression trees.

According to the number of attributes evaluated in each test condition – i.e., each internal node – two DT types can be induced: *univariate* and *multivariate* [30]. In the former, each test condition evaluates a single attribute to split the training set, while a combination of attributes is used in multivariate DTs. Some of the advantages of univariate DTs are their comprehensibility and the simplicity of their induction algorithms; however, they may include many internal nodes when the training data set instance distribution is complex. On the other hand, multivariate DTs classifiers commonly show better performance, and they are smaller than univariate ones. Nevertheless, they are less expressive and require more computational effort to induce them.

| Error (%) | Reward (%) | Stride 0 | Stride 1 | Stride 2 | Stride 3 | Stride 4 | Stride 5 | Stride 6 | Stride 7 | Stride 8 |
|---|---|---|---|---|---|---|---|---|---|---|
| 86.54 | 88.44 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 6 | 7 |
| 46.48 | 75.14 | 1 | 2 | 1 | 0 | 0 | 1 | 0 | 4 | 2 |
| 26.87 | 38.72 | 3 | 1 | 3 | 0 | 0 | 2 | 0 | 2 | 0 |

Table 6.13: Inner-loops approximation using the Loop2 operator.

In univariate DTs, test conditions are defined as $x_i \lessgtr c$, where $x_i$ is the i-th attribute value, and $c$ is a threshold value used to define a partition. Therefore, test conditions represent axis-parallel hyper planes dividing the instance-space, so they are also known as axis-parallel DTs. Anyway, when a categorical attribute is evaluated, the training set is split into as many subsets as values exist in the attribute domain.

In the case of a linear combination of attributes, such in Equation (6.29), DTs are named oblique, as their test conditions represent hyper planes having an oblique orientation relative to the instance-space axes. Finally, whether non-linear combinations of attributes are employed, DTs referred to as non-linear.

$$\sum_{i=0}^{d} w_i x_i \lessgtr c \tag{6.29}$$

## 6.2.1 The tree-construction problem

In its essence, the tree-construction problem consists in using the training data set – that is constituted of historical, labeled data – to determine its best splits. These splits reduce the data set into smaller and smaller pieces, while aiming at splits that best emphasizes the differences between data points belonging to the different partitions. The most widely adopted algorithms to construct DTs are CART and C4.5.

### 6.2.1.1 Classification and Regression Trees: the CART algorithm

The CART algorithm [30] is a recursive binary-partitioning procedure that is capable of handling both continuous and categorical attributes. A complete discussion of the algorithm is undoubtedly burdensome and long, since there are multiple splitting rule for both classification and regression predictors, separate handling of numerical and categorical attributes, provision for missing values and so forth. Anyway, we report a

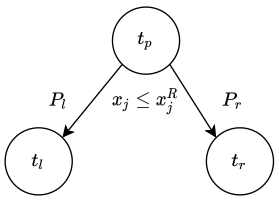



Figure 6.19: Example of node-splitting in DTs

brief and concise discussion of the method.

CART begins at the root node, i.e., the node including the whole training data set, by searching the data for the best splitter available, i.e., by testing each attribute-value pair for its goodness-of-split. The splitting process is repeated producing grand-children of the root node, and so forth, until no further split is possible. No stopping rule is adopted, and the resulting *maximum-size tree* is, then, pruned back to the root, removing subtrees which contribute the least to the overall performance.

According to [30], binary splitting should be preferred against multi-way splitting, since it fragments the data set slowly w.r.t. multi-way splitting, and it allows multiple splits on the same attribute, which may increase the predictive performance. Moreover, since CART does not discard any of the data points, optimal splits are always invariant w.r.t. order-preserving transformations of attributes, including logarithmic, square-root and power transformations.

The effectiveness and expressiveness of a tree are significantly affected by the adopted splitting criterion. CART splitting rule are always expressed in the form "an instance goes left if *condition* is true, otherwise it goes right". *condition* is expressed either in the $x_j \leq x_j^R$ for continuous numerical attributes, or as $x_j \in \left\{x_j^1, \cdots, x_j^n\right\}$ for categorical values. Here, $x_j$ is an attribute, $x_j^R$ its threshold value for numerical attributes, and $\left\{x_j^1, \cdots, x_j^n\right\}$ is a list of values for $x_j$ if $x_j$ is a categorical attribute. Consider a parent node $t_p$ and its left and right child nodes $t_l$ and $t_r$, as depicted in Figure 6.19. The CART algorithm identifies the attribute, and its value, that splits the data in two portions, with maximum homogeneity. The latter is defined using the *impurity function* $i(t)$, and it is achieved by maximizing the *change of impurity function* in Equation (6.30), where $t_p$ is the parent node, $t_c$ the set of child nodes, and $E$ is the expected value.

$$\Delta i(t) = i(t_p) - E\left[i(t_c)\right] \tag{6.30}$$

Assuming $P_l$ and $P_r$ the probabilities of a data point to follow the left or the right node, respectively, the $\Delta i(t)$ function can be rewritten as follows.

$$\Delta i(t) = i(t_p) - P_l \cdot i(t_l) - P_r \cdot i(t_r) \tag{6.31}$$

Therefore, for each node, the CART has to solve the optimization problem in Equation (6.32), which implies the CART has to search through all possible values of all attributes for the best split, i.e., the split that maximizes $\Delta i(t)$.

$$\underset{x_j \leq x_j^R}{\arg\max} \left\{ i(t_p) - P_l \cdot i(t_l) - P_r \cdot i(t_r) \right\} \tag{6.32}$$

Author of [30] discussed several definitions for the impurity function. Two of the most commonly adopted definitions are briefly discussed below.

**6.2.1.1.1 The Gini impurity** Concerning classification trees, the Gini impurity, which is named after the Italian mathematician Corrado Gini, is the most commonly adopted impurity definition. It is reported in Equation (6.33), there $k$ and $l$ are class indexes, and $p(k|t)$ and $p(l|t)$ are the conditional probability of class $k$, or $l$, provided the current node is $t$.

$$i(t) = \sum_{k \neq l} p(k|t) \cdot p(l|t) \tag{6.33}$$

Consider $k \in [1, \cdots, K]$, applying the Gini impurity to Equation (6.31) results in the following.

$$\Delta i(t) = -\sum_{k=1}^{K} p(k|t_p)^2 + P_l \sum_{k=1}^{K} p^2(k|t_l) + P_r \sum_{k=1}^{K} p^2(k|t_r) \tag{6.34}$$

Therefore, for each node, the CART has to solve the optimization problem in Equation (6.35).

$$\underset{x_j \leq x_j^R}{\arg\max} \left\{ -\sum_{k=1}^{K} p(k|t_p)^2 + P_l \sum_{k=1}^{K} p^2(k|t_l) + P_r \sum_{k=1}^{K} p^2(k|t_r) \right\} \tag{6.35}$$

**6.2.1.1.2 The towing impurity** Another commonly adopted impurity definition is the towing impurity that, unlike the Gini one, searches for two classes which make up together more than 50% of the data, defining the change of impurity as follows.

$$\Delta i(t) = \frac{P_l \cdot P_r}{4} \cdot \left( \sum_{k=1}^{K} |p(k|t_l) - p(k|t_r)| \right)^2 \tag{6.36}$$

Then the CART must solve the optimization problem in Equation (6.37). Although it results in more balanced trees, the towing impurity is slower w.r.t the Gini one.

$$\underset{x_j \leq x_j^R}{\arg\max} \left\{ \frac{P_l \cdot P_r}{4} \cdot \left( \sum_{k=1}^{K} |p(k|t_l) - p(k|t_r)| \right)^2 \right\} \tag{6.37}$$

### 6.2.1.2 The C4.5 algorithm

The C4.5 is not a single algorithm, rather a suite of algorithms with different features [143]. It is named after the ID3 approach [142] for tree induction.

All algorithms part of the C4.5 suite begin with a root node that represents the entire given data set, which is recursively split into smaller and smaller subsets by testing for a give attribute at a time. C4.5 adopts information-theory based splitting criteria, such as the *gain* – i.e., the reduction in entropy of the class distribution due to applying a test – and *gain-ratio* – which overcomes the tendency of the gain criterion to favor multi-way tests. The goal of the algorithm is to select the right attribute, and the

corresponding value, so that the entropy of class distribution is reduced. The spitting procedure continues until subsets are *pure*, i.e., all instances in a subset fall into the same class. C4.5 is not restricted to considering only binary tests, rather it allows tests with more outcomes. If the attribute type is Boolean, the possible outcomes are only two, of course, but test for categorical attributes can be multivalued. Conversely, for numerical attributes, tests can be only binary-valued, and expressed as in the CART algorithm, i.e., as $x_j \leq x_j^R$.

At the end of the splitting procedure, the resulting tree is pruned, in order to reduce its complexity. The C4.5 algorithm adopts a peculiar pruning technique, namely the *pessimistic pruning*. This technique does not require a test data set, rather it estimates the error that might occur based on the amount of misclassifications in the training data set. The approach recursively estimate the error-rate associated with a certain node based on the estimated error-rate of child nodes. Consider a leaf node with $N$ correctly predicted instances and $E$ instances that do not belong to the class corresponding to the considered leaf: the pessimistic technique determines the empirical error-rate at the leaf node as follows.

$$\frac{E + \frac{1}{2}}{N} = \frac{2E + 1}{2N} \tag{6.38}$$

For a subtree having L leaves, $\Sigma N$ correct prediction and $\Sigma E$ erroneous predictions, the error rate is estimated as follows.

$$\frac{\Sigma E + \frac{L}{2}}{\Sigma N} = \frac{2\Sigma E + L}{2\Sigma N} \tag{6.39}$$

Now, suppose a subtree is suppressed using the best leaf node, i.e., the leaf node exhibiting the lowest error-rate, introducing $J$ misclassifications, the pessimistic pruning will actually replace the considered subtree using the pinpointed leaf i.f.f. Equation (6.40) is satisfied. There, $\sigma(\cdot)$ denotes the standard deviation.

$$J + \frac{1}{2} \leq \sigma(E + \frac{L}{2}) \tag{6.40}$$

The approach discussed above can be extended to prune based on confidence intervals, modeling the error-rate at leaves as a Bernoulli random-variable. Thus, for a given confidence threshold $c$, an upper bound $e_{max}$ for the error-rate such that $P(e < e_{max}) = 1 - c$ can be determined. Furthermore, if $N$ – i.e., the amount of correctly predicted instances – is large, the error-rate distribution can be approximate using a zero-mean-unitary-variance normal distribution $\mathcal{N}(0, 1)$. In this case, an upper-bound $e_{max}$ for the error-rate can be determined using Equation (6.41), where $z$ is selected according to the desired confidence interval.

$$e_{max} = \frac{e + \frac{z^2}{2N} + z\sqrt{\frac{e}{N} - \frac{e^2}{N} + \frac{z^2}{4N}}}{1 + \frac{z^2}{N}} \tag{6.41}$$

### 6.2.2 Bagging predictors

Bagging, that stands for **b**ootstrap **agg**regat**ing**, is a method for generating multiple version of a predictor – not necessarily a DT – and using these to get an aggregate predictor [31].

Consider a training data set $T = \{(\mathbf{x}_i, y_i)\ \ i = 1 \cdots N\}$, where $\mathbf{x}_i$ is the vector of input features and $y_i$ either a class label or a numerical response. Assume there is a procedure, such as the CART or C4.5, for using this training data set to construct a predictor $\varphi(\mathbf{x}, T)$. Using this predictor, we can predict the class $y$ to which $\mathbf{x}$ belongs to. Now, suppose we have a sequence $\mathbb{T} = \{T_1, \cdots, T_k\}$ of training data sets, each consisting of $N$ independent observation taken from the same underlying distribution as the above-mentioned $T$. The sequence $\mathbb{T}$ can be exploited to get a better predictor than $\varphi(\mathbf{x}, T)$, working with the set of predictors $\{\varphi(\mathbf{x}, T_1), \cdots, \varphi(\mathbf{x}, T_k)\}$. If predictions are numerical, then the aggregate predictor outcome can be obtained by averaging over $\{\varphi(\mathbf{x}, T_i)\}$, i.e.,

$$\varphi_A(\mathbf{x}) = E_T\{\varphi(\mathbf{x}, T)\} = E_T\{\varphi(\mathbf{x}, T_i), i = 1, \cdots, k\} \tag{6.42}$$

where $\varphi_A(\mathbf{x})$ denotes the aggregate predictor and $E_T\{\varphi(\mathbf{x}, T)\}$ the expectation over $T$. Conversely, if predictions are categorical, then, a method for aggregating $\{\varphi(\mathbf{x}, T_i)\}$

is by majority-voting, i.e.,

$$\varphi_A(\mathbf{x}) = \arg\max_j \; nr\{k : \varphi(\mathbf{x}, T_i) = j\} \tag{6.43}$$

Anyway, learning involves almost always a single training data set. Nevertheless, an imitation of the process leading to $\varphi_A(\mathbf{x})$ can be defined through bootstrap-sampling [65] a set of training data sets $\left\{T_i^B,\ i = 1 \cdots k\right\}$ from T, to form $\left\{\varphi_A\left(\mathbf{x}, T_i^B\right),\ i = 1 \cdots k\right\}$. The $\left\{T_i^B,\ i = 1 \cdots k\right\}$ set defines multiple data sets, each drawn from the bootstrap-distribution approximating the one underlying $T$. Each data set consists of $N$ samples extracted randomly from $T$ while using replacement, which means each $(\mathbf{x}_i, y_i) \in T$ may appear either more than once or not at all in $\left\{T_i^B\right\}$. If changes in $T$ produces substantial changes in $\varphi(\mathbf{x}, T)$, i.e., if the induction procedure is unstable, bagging can give substantial gains in terms of classification accuracy, as proven in [31]. This is because the bootstrapping procedure builds uncorrelated data sets, which tend to decrease the variance of the model without affecting its bias. As a result, while a single predictor may be quite sensitive to noise, the average of many predictors is not, as long as predictors are uncorrelated.

### 6.2.3 From bagging to Random-forest predictors

Significant improvements in classification accuracy have resulted from growing an ensemble of trees and letting them vote for the most popular class. Besides bagging, several other techniques have been proposed. Some examples are random split selection [62], where the split at each node is randomly selected among $k$ different "best splits", random feature selection, which either does a random selection of a subset of features to be used to grow trees [82], or searches for over a random selection of features for the best split at each node [16], and random error injection [33].

According to [34], all these procedures generate random forest classifiers. Extrapolating these techniques to the maximum, in fact, in order to construct the k-th tree, all the mentioned procedures generate a random vector $\Theta_k$, which must be independent w.r.t. $\Theta_1, \cdots, \Theta_{k-1}$ yet drawn from the same distribution, and a tree is grown using both the training set and $\Theta_k$, resulting in a classifier $\varphi(\mathbf{x}, \Theta_k)$. The nature and the dimensionality of $\Theta_k$ depend on its use during the construction procedure. In bagging,

for instance, $\Theta_k$ is used to extract $T_i^B$ from $T$, while in random feature selection it is used to randomly chose a feature, or a subset of features.

Random forest classifiers allow a consistently lower error w.r.t. other construction methodologies, but still exhibit lower accuracy w.r.t. other algorithms performing adaptive reweighing of training set [68]. Therefore, the author of [34] propose to inject further randomness in order to minimize the correlation between predictors while keeping their strength unaffected. To do so, he combined bagging with random feature selection: each new training set is drawn, with replacement, from the original training set, and random feature selection is adopted while growing each tree, without using any pruning technique.

The adoption of bagging provides two relevant advantages: it allows enhancing accuracy, of course, and it allows constructing out-of-bag classifiers. The latter allow estimating the generalization error of the combined ensemble of trees. Assume a method for constructing a classifier from a training set $T$, form bootstrap training sets $T_k, k = 1 \cdots K$, and construct $\varphi(\mathbf{x}, \Theta_k), k = 1 \cdots K$, letting the latter vote to form a bagged predictor. For each $(\mathbf{x}, y) \in T$, an out-of-bag classifier aggregates the outcome of $\varphi(\mathbf{x}, \Theta_k), k = 1 \cdots K$ i.f.f. $(\mathbf{x}, y) \notin T_k$. Studies on bagged classifier empirically proved that the error-estimation provided by out-of-bag predictor is as accurate as the estimation obtained by using a test set of the same size of the training set [32].

### 6.2.4 Hardware accelerators targeting decision-tree based classifiers

As mentioned, their simplicity and understand-ability make DTs one of the most popular machine-learning algorithms [179]. Though, at the beginning, they were not considered for hardware accelerator, since they need only comparison to be performed during the inference phase; thus, they were not viewed as to be computationally expensive. Nevertheless, the need to accelerate the inference phase emerged inherently in some application fields. Many real-time tasks require high prediction speed. However, the software implementation of the inference phase cannot meet the requirement even if the multi-threading technology is adopted. Hence, attention has been paid to the hardware-based accelerators [95].

Authors of [168] compared performances of different implementations of hardware accelerators for DT-based predictors while using off-line generated models. They empirically proved that FPGA implementations provide the highest performance, but may require a multi-chip / multi-board system to execute even modest sized predictors. GP-GPU implementations offer a more flexible solution with reasonably high performance that scales with predictor size. Finally, multi-threading via OpenMP on a shared memory system, which is the simplest solution, is able to provide near linear performance that scaled with core count, but was still significantly slower than the GP-GPU and FPGA implementations. This is because classification has an inherently highly parallelism, since each decision tree processes every sample independently, and the only synchronization occurring when the results of all the decision tree are combined to provide a final classification for a sample. However, when the decision trees within the forest vary significantly in terms of shape and depth, it is challenging to apply pipelining and SIMD / SIMT parallelization techniques, since the time to process a sample is data dependent. Additionally, this irregularity in tree size and shape makes it difficult to provide deterministic memory access into the tree. Furthermore, the presence of very deep trees within the forest makes it prohibitively expensive to apply techniques that improve regularity, such as fully populating all trees so that the processing time for each sample is identical [168, 163, 36].

Authors of [132] further investigate on GP-GPUs and pinpointed three main reasons why the GP-GPU is an ill-fated choice for accelerating DT-based predictors. First, in order to perform the inference phase, each node in the tree is evaluated using an *if-then-else* statements, which compare input values with constant values, both represented by a floating-point representation. Although the GP-GPU supports a double precision floating-point, such highly precision might not be required for the classification. Therefore, a high-precision arithmetic circuit is inefficient both the amount of hardware and power consumption. Second, GP-GPU cores are specialized in data parallel computation. However, the DT-based predictors may consist of trees with a different size and depth, that causes an unbalanced computation. Hence, the computation-time would be bound by the tree which has the longest path. Last, but not least, communication between near processing cores with the same local memory can be performed at a relatively high speed, while communication-penalty is large for the all-to-all communications. Since a DT-based predictor requires the whole of the classification after evaluating all the trees, the all-to-all communication would always

occur, which is certainly a performance bottleneck.

According to [111], architectures of FPGA-based accelerators for DT-based predictors can be categorized as comparator-centric and memory-centric. The former implement the model as a threshold network that consists of a layer of threshold logic units and a layer of combinational logic units, while the latter accelerate the model by introducing the pipeline in layers of the tree.

#### 6.2.4.1 Memory-centric architectures

Figure 6.20 depicts the RTL schema of a memory-centric accelerator. Each of the trees has $N$ "internal " levels and a final "leaves level", but if a leaf resides at level $j \leq N$, then the tree is expanded converting the leaf to an internal node, with identical child nodes. Each of the latter has the same label as its parent, and also bypass the comparator. This allows to regularize the architecture, even if trees are unbalanced.

Figure 6.21 depicts the RTL schema of an internal level. The attribute-value pairs to be compared against the data sample are stored in memory. Since each data sample being classified can traverse a single node of the tree at a given level, a single memory-read operation is required. The current memory-address is used to select the appropriate attribute-value pair from the memory block, and the comparator outcome is, then, exploited to construct the memory-address for the subsequent level. Finally, leaves levels consist of a look-up table containing class labels.

The whole architecture stands out for its flexibility, since it can be adapted to a large variety of models [153, 163, 136, 66, 10]. In addition, in order to update the implemented model, the designer only has to update each memory block. Moreover, it allows satisfying high throughput demands, since pipelining mechanisms are straightforward to implement. On the other hand, this architecture has several drawbacks. First, trees which do not reach the maximum-depth are expanded in order to allow synchronization, wasting resources and increasing the power consumption.

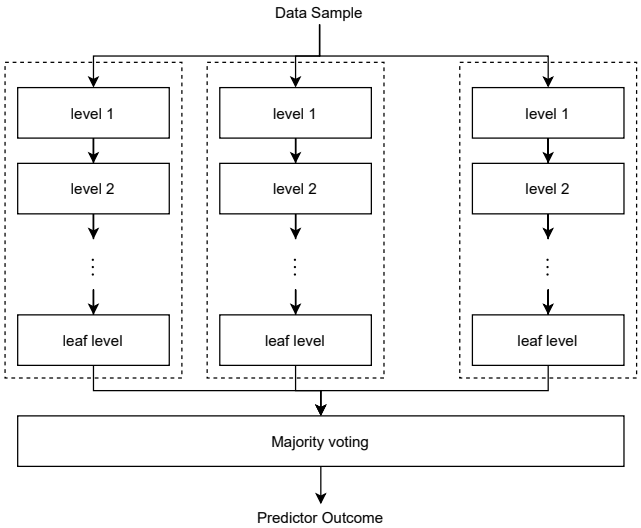



Figure 6.20: RTL architecture of a memory-centric accelerator.

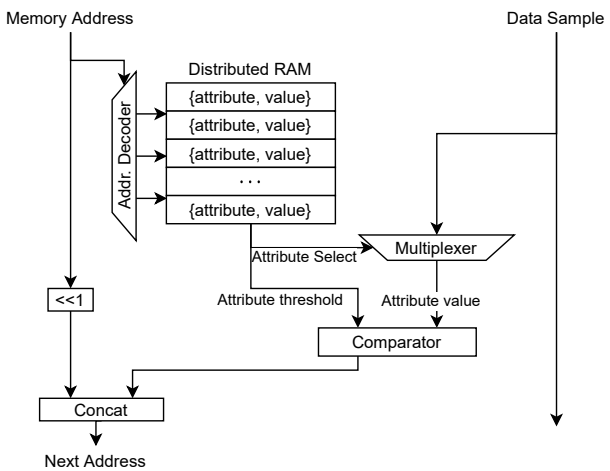



Figure 6.21: RTL schema of a memory-centric accelerator cell.

#### 6.2.4.2 Comparator-based architectures

The overall architecture of comparator-based accelerators resembles the memory-centric one. It reduces the reliance on memory elements by using custom comparators at each internal level. This eliminates the ability to quickly update parameters, but benefit lower area requirements. Conceptually, a custom comparator is a direct representation of an internal node, which is possible only if it assumes the comparison attribute and comparison value are constants. The data flow is similar to the memory-centric architecture: each internal level i processes data sample d in the same way as in the memory-centric architecture, but with addresses of custom comparators instead of memory elements. This architecture has been successfully adopted in a large variety of classification problems [57, 89].

Comparator-centric accelerators could achieve high throughput and low latency as well; however, since designing such accelerator requires all information of the model for the logic design, the architecture is tightly coupled with the underlying model, meaning any change in the model requires the accelerator to be reconfigured. For application scenarios where the model updates frequently, the application of such accelerators will undoubtedly introduce significant maintenance costs.

#### 6.2.4.3 A custom processor for accelerating Decision-Treee-based predictors

In order to overcome limitations of both the mentioned architectures, authors of [159] proposed RF-RISA, a dedicated processor, inspired by the Reduced Instruction-Set Computer (RISC) architecture. Authors claim it decouples the parameters of the model from its hardware implementation, employing a set of instructions to encode the nodes and structure information of the model, hence avoiding hard-coding the parameters into the hardware. A software driver encodes the structure and node information of the model into a group of instructions. Meanwhile, a novel hardware architecture is designed to provide storage resources and support pipelined computing. Then, the instructions are dynamically mapped into the memory, and executed in parallel. All parameters of the model are transparent to the hardware until the instructions are executed in RF-RISA.

The instruction set encodes the model: it consists of three types of instructions. Node-operation instructions encode the information of an internal node, including its relative address, the attribute being compared and the corresponding threshold, the left and right child-node types, and the left and right child-node information. If the left or right child-node type is “0”, then the corresponding node is a leaf, and the corresponding label is stored in the appropriate child-node information field of the instruction. Conversely, if the node is an actual internal node, the relative address of the child-node is stored into the corresponding information field of the instruction. First-layer instructions are encoded as a bit-vector. A “1” in such bit vector indicates that the corresponding layer consist only of the root node of a tree. Finally, control instructions control how node-operation instruction and first-layer instructions are mapped to the FPGA. When the software driver configuring the architecture fetches a control-instruction, if its type is “node-operation” then its “memory-address” field dictates where to load all subsequent node-operation instructions. Thereafter, all subsequent instructions are loader to the FPGA until a new control-instruction is fetched. On the other hand, if the type of the fetched control-instruction is "first-layer", then the subsequent first-layer instruction, and the bit-vector it contains, are loaded into the configuration registers.

A RTL-schema of the accelerator, along with the model it implements, is reported in Figure 6.22: $m$ RAM blocks are organized as logically connected units to store the node-operation instructions. If one block is inadequate for storing the instructions of one layer, the latter instructions can be placed in consecutive blocks. The node-operation instruction of the same layer of a tree are stored in memory blocks in order from left to right, and each memory block only stores the instructions of the same layer. Each RAM block is connected to a processing element, that execute the instruction, and buffers are set to store the intermediate results.

To predict an instance, RF-RISA starts its execution from the first node-operation instruction stored in BRAM 0. If the comparison result of the instruction points to an internal node, the location of the next instruction of the node is obtained, and subsequent memory blocks are accessed; otherwise, the current instruction output a class-label. The prediction is finished after all memory-blocks are accessed, and the prediction result is given by the majority voting. One prediction per clock cycle can be achieved, with prediction-latency of $m + 1$ clock cycles.

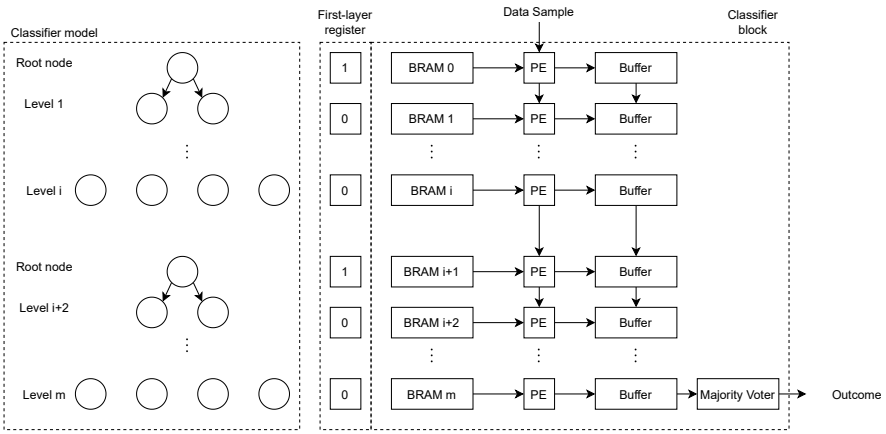



Figure 6.22: Hardware architecture of the RF-RISA accelerator from [159]

This architecture differs from the memory-centric architecture depicted in Figure 6.20 in the way how the memory and processing elements are organized. In facts, the memory-centric architecture groups memory blocks into stages, which correspond to layers in the tree. The sizes of stages are determined in advance, and each stage shares one processing element, By contrast, in RF-RISA each memory block is an independent unit and owns a dedicated processing element. Moreover, $m$ is the only parameter that needs to be set in advance. Nevertheless, the decoupled nature of the implemented model and the fact that a single parameter governs the entire architecture may be a major limitation in some applications. Moreover, the general scheme of the architecture leaves little room for the introduction of approximation.

#### 6.2.4.4 Speculative architectures

The approach proposed in [15, 14, 21] to implement random-forest classifier in hardware is quite interesting. In order to speed-up DTs visiting, it adopts a speculative approach, which consists in a DT flattening so that the visiting is performed over every possible path. In particular, each DT node contains a condition that establishes if the visiting has to continue on left subtree or on right subtree, until a leaf is reached. Instead, in the speculative approach, predicates are performed concurrently, regardless of the position and depth at which nodes are located: a Boolean decision variable, that

indicates whether a condition is fulfilled, is produced for each one of the evaluated predicates. In order to determine which leaf of the DT is reached, i.e., which class the input belongs to, a Boolean function, called *assertion*, is defined for each different class. Since a path that leads to a specific leaf is obtained by computing the logic-AND between the Boolean decision variables along that path, and since it is possible to compute the logic OR between the conditions related to different paths leading to leaves belonging to the same class, assertions can be defined as a sum of products Boolean functions.

For the sake of clarity, let us consider the DT depicted in Figure 6.23, which evaluates two features in order to assess which one of three classes the inputs belong to. Starting from the root node, descending the DT and visiting nodes from the left to the right, the Boolean decision variables involved in the classification process are Q1, which is produced at the root node, Q2 produced at the $f_2 < 10.9$ node, and so on. Let us consider the $\alpha$ class: an input vector belongs to it if $f_1 \geq 4$ – Q1 is false – **and** $f_2 \geq 10.9$ – Q2 is false – **or** $f_1 < 4$ – Q1 is true – **and** $f_2 \geq 27.5$ – Q3 is false – **and** $f_1 < 17$ – Q4 is true. In Equation 6.44 we report Boolean assertions for all the classes.

$$\begin{aligned} \alpha &= (\overline{Q1} \wedge \overline{Q2}) \vee (Q1 \wedge \overline{Q3} \wedge Q4) \\ \beta &= (\overline{Q1} \wedge Q2) \vee (Q1 \wedge Q3 \wedge Q5) \vee (Q1 \wedge \overline{Q3} \wedge \overline{Q4}) \\ \gamma &= Q1 \wedge Q3 \wedge \overline{Q5} \end{aligned} \tag{6.44}$$

Predicates are evaluated using Decision-BoXs (DBXs), i.e., comparators, while the visiting algorithm can be performed as a multi-output Boolean function. A comprehensive block schema is depicted in Figure 6.24.

This architecture stands out for its regularity and flexibility, so that it can be automatically generated starting from the PMML encoding of the model, as shown in [15, 14]. Moreover, the scalability of this approach has been formally demonstrated in [20]. In particular, the number of literals in each assertion is always less or equal to twice the size of the features set.

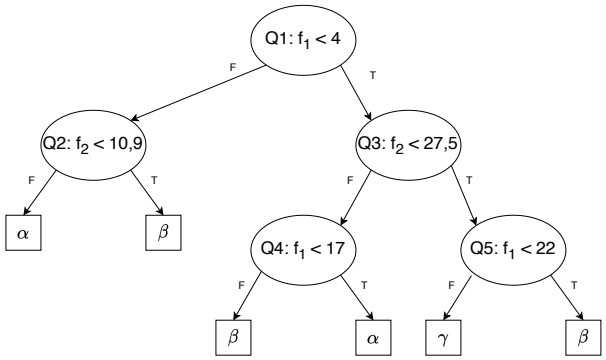



Figure 6.23: An example of decision tree.

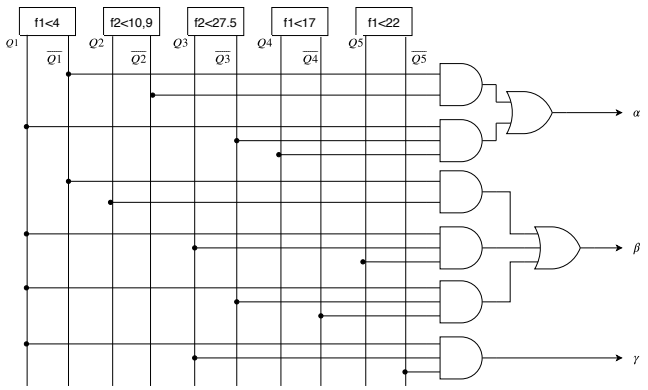



Figure 6.24: Hardware implementation of a decision tree.

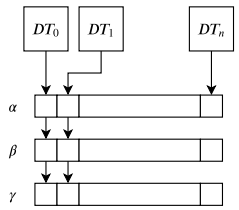



Figure 6.25: Decision vectors.

**6.2.4.4.1 Implementing majority-voting in hardware** The outcome of each assertion-function corresponding to a certain class, but computed by a different DT are arranged in an array of *N* elements, as depicted in Figure 6.25, with *N* being the number of DTs. A majority voter is used to state which class is the winner.

Let $d_{i,j}$ be the preference expressed by the $i-th$ DT for the $j-th$ class, i.e., $d_{i,j}$ is a Boolean variable being equal to 1 i.f.f. the classifier input has been recognized by the $i-th$ DT to belong to the $j-th$ class; the following matrix can be defined:

$$\mathbb{D} = \begin{bmatrix} d_{0,0} & d_{0,1} & \cdots & d_{0,M-1} \\ d_{1,0} & d_{1,1} & \cdots & d_{1,M-1} \\ \vdots & \vdots & \ddots & \vdots \\ d_{N-1,0} & d_{N-1,1} & \cdots & d_{N-1,M-1} \end{bmatrix} \tag{6.45}$$

We define $p_j = \sum_{i=0}^{N-1} d_{i,j},\ 0 \leq j < M$. Since each DT expresses just one preference (i.e., $\sum_{j=0}^{M-1} d_{i,j} = 1\ 0 \leq i < N$), it follows that the class $w$ is the most voted i.f.f. $p_w > p_j,\ \forall j \neq w$, while we get a draw condition i.f.f. $\exists\{i,j\}$ s.t. $p_i = p_j = max_{0 \leq k < M}\{p_k\}$.

Rather than using binary adders to state which class gets the highest score, the majority voter sorts each column of the matrix $\mathbb{D}$ using a parallel sorting algorithm, pretty much like bubble-sort, by shifting all the high bits at the beginning of each column. This process is performed by exploiting a Boolean circuit called the *sorting network*, which depth is equal to $n$.

Let us consider a two bits array: Table 6.14 reports the truth table of a two bits

| $x_1$ | $x_0$ | $y_1$ | $y_0$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Table 6.14: Truth table of a 1 bit sorting network

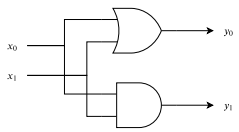



Figure 6.26: A single sorter-cell.

sorting network. It is easy to recognize that $y_0 = x_0 \vee x_1$ and $y_1 = x_0 \wedge x_1$. Conversely, defining *n*-bits sorting network is cumbersome. However, such a network can be built using multiple two-bits sorting networks arranged in a *n*-stages pipeline, with even stages consisting in $N/2$ two-bits sorters – each of which compares array elements starting from even positions – and odd stages consisting in $N/2 - 1$ two-bits sorters – each of which compares array elements starting from odd positions [21]. The sorting networks need at least $N/2$ clock cycles to provide sorted arrays. An example of such a network is provided by Figure 6.27, while Figure 6.26 depicts a single sorting-cell.

Once the votes are sorted, the score each class have received needs to be verified. Let us define a threshold indicator as follows:

$$\tau_{i,j} = \begin{cases} 1 & i \leq p_j \\ 0 & i > p_j \end{cases}, \; 2 \leq i \leq N/2 \tag{6.46}$$

Hence, to detect the most voted class we need to find the $\tau_{i,j} = 1$ with the highest $i$: in

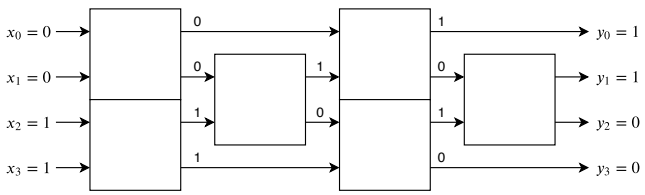



Figure 6.27: Four-bits sorting network.

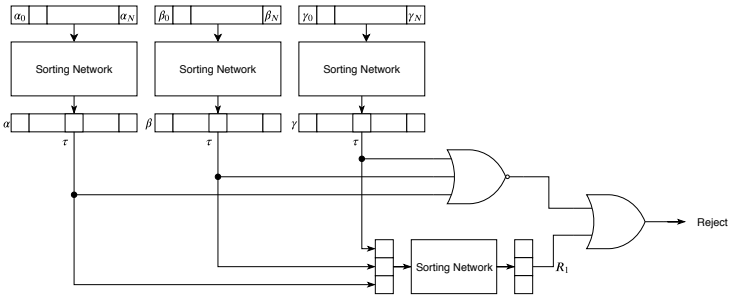



Figure 6.28: Detailed block schema of the rejection module.

case it is unique, we get a voted class, otherwise we got a draw. Exploiting the same sorting network used before, we can easily detect these two conditions.

Figure 6.28 shows an example of such a module for a three-class classifier.

### 6.2.5 Approximate DTMCSs

Concerning DT MCSs, the wide-spread adoption of hardware-based accelerators is actually hindered by scalability issue, as reported in [168]. Nevertheless, there is very little research on both efficient architectures and methods to reduce their hardware-resource consumption, as the effort is mainly devoted to other classification systems, e.g., DNNs.

One of the few contributions from the scientific literature concerning the improvement of energy-efficiency of DT MCSs has been proposed in [173]. Authors noticed that, usually, only a part of the data of a given data set really needs the full computational power of a classifier. Therefore, they dynamically configure the classifier making it more or less accurate, according to the difficulty in classifying the inputs. Hence, rather than building a single complex model, during the training phase they construct a set of models with progressively increasing complexity. Then, during the testing phase, the number of decision models applied to a given input varies depending on the difficulty of the considered input instance. In order to estimate the difficulty of a certain input, a confidence level for each classification is computed. If the latter confidence falls above a certain threshold, the classification process is terminated, otherwise, a

more accurate classifier is used.

In [23], authors replaces exact comparators using approximate ones, designed using the precision-scaling technique, in order to reduce the hardware overhead of DT MCSs. They use the Branch & Bound (B&B) algorithm to introduce approximation while the constraining the classification-accuracy loss. Despite numerous improvements that the authors have made to the B&B implementation, such as pre-pruning of the tree and grouping features by information gain, they have managed to evaluate only a few classifiers, and for each of them only a few approximate configurations. This has greatly limited the quality of the solutions obtained.

### 6.2.6 Applying the methodology to DTMCSs applications

In this Section we discuss case-studies concerning the application of our methodology to DT MCSs.

As will be detailed in the following, in order to generate approximate variants and to perform DSE, we again resort to the E-𝕀DE𝔸 framework, that we extensively discussed in Section 5.1. Hence, through a MOP-based DSE, we find the correct approximation degrees leading to non-dominated solutions exhibiting near-Pareto trade-offs between accuracy-loss and hardware-efficiency. As for case-studies discussed above, E-𝕀DE𝔸 allows specifying multiple fitness-functions; hence, for our scenario, we define the accuracy-loss and the hardware requirements to be both minimized.

Resorting to Figure 6.3, given a DT MCS implementation, the first step is approximate variant generation, which is performed by exploiting the Clang-Chimera tool. Exploiting the regularity and simplicity of DT MCSs, we generate C++ implementations straight from the PMML. The approximate variants resulting from Clang-Chimera are fed to the Bellerophon DSE engine: approximate configurations resulting from assigning a value to each configuration parameter within variants are evaluated in terms of fitness-functions, and Pareto-optimal ones are selected using the NSGA-II heuristic. At the end of the DSE, the latter configurations are exploited to configure the accelerator we discussed in Section 6.2.4.4, in order to assess the actual hardware requirements through FPGA synthesis and simulations.

#### 6.2.6.1 Generating approximate variants

The scientific literature is not the only source to draw while identifying approximable parts of a given application. Indeed, the architecture of the final hardware accelerator may provide undoubtedly useful suggestions in order to effectively select parts of the model to approximate, hence a suitable approximation technique.

Several opportunities inherently arise, for instance, from the hardware implementation which is discussed in Section 6.2.4.4. Both comparators and assertion-functions are quite good candidates for approximation, albeit the contributions of the former seem much more substantial w.r.t the latter, meaning their approximation can lead to significant savings. Furthermore, in order to introduce approximation, a wide plethora of techniques can be used with respect to comparators, including precision-scaling, inexact hardware, and functional approximation. Since the latter is well suited to Boolean functions, it is also ideal for introducing approximation within assertion-functions.

As mentioned, DT MCSs may consist of hundreds, or even thousands of nodes, each comparing one of the many features taken into account by the predictive model with their corresponding threshold value. Consider introducing approximation through approximate comparators designed using the precision-scaling technique: such approximate comparators allow, by means of a configuration parameter, to govern the degree of introduced approximation by tuning the amount of neglected mantissa bits ad each comparison. Hence, the value to be assigned to each one of such parameters constitute the decision variables of the MOP. Furthermore, the domain of such variables depends on the particular data-type being adopted for representing features. However, albeit straightforward, this naive MOP definition may result in an utterly infeasible DSE. Indeed, hardware implementations of DT MCSs may consist of hundreds, or even thousands of comparators, resulting in an enormous design-space. Nevertheless, we can exploit the fact that, albeit against different threshold values, the same feature can be taken into consideration for comparison several times while visiting trees. This allows to set the same degree of approximation to all the comparators processing the same feature, reducing the number of decision variables, and, consequently, the size of the solution space. Thus, being $F$ the features set, each approximate configuration – i.e., individual in the GA context – can be represented using a vector consisting of $|F|$ elements – i.e., genes –, each governing the degree of approximation for the

corresponding feature.

As we mentioned, the actual generation of approximate variants is performed while resorting to the E-𝕀DE𝔸 framework. The regularity and simplicity of DT MCSs allow us generating C/C++ implementations straight from the PMML encoding of the model. The latter encoding resorts to the double-precision floating-point representation for input and thresholds. Hence, we generate variants exploiting the FLAP mutator provided by Clang-Chimera, which allow applying the precision-scaling technique on comparators when using the mentioned double-precision floating-point representation.

#### 6.2.6.2 Design space exploration

For what pertains to hardware-requirements, in order to accurately take into account the resource savings in the DSE, we should measure area, power consumption and maximum clock speed of the explored approximate variants. This would require the hardware synthesis – and also simulations, in the case we are interested in measuring power consumption, for instance – of each variant explored in the DSE. This is utterly a time-consuming process. Hence, again, we resort to a model-based gain estimation to drive the DSE.

Since the purpose of the model is determining the $<$, $=$ or $>$ relation between two different approximate configurations, it is not necessary to focus on its accuracy. We rather focus on its fidelity, i.e., how often the estimated values are in the same relation as the real values for each pair of configurations. Concerning comparators, it is easy to recognize the fewer bit to compare, the fewer hardware requirements.

#### 6.2.6.3 Case-study #1: validating the method

To prove the robustness of the proposed methodology, we exploited PMMLGen tool [20] to provide different workloads, in terms of models, to be approximated.

In particular, we collected 51 different datasets varying on the number of features (from 1 to 50) and number of classes (from 2 to 20). Then, we trained, for each data set, a single DT classification model and random-forest classification models with different

number of DTs, namely 5, 10, 15 and 20. For each of the 255 trained classifiers, we found several approximate solutions by means of approximate exploration. Then, we synthesized the ones that belong to the Pareto-frontier bounds, i.e., the ones characterized by best reward value, meant maximum reduction of area overhead, and ones affected by minimum accuracy loss. We report the amount of resource-occupation gain (in terms of FPGA LUTs and registers) and the accuracy loss evaluated in percentage w.r.t. the original synthesized model in Figure 6.29 and 6.30, respectively for maximum area overhead reduction and minimum accuracy loss synthesized solutions. Please, kindly note that, although both in percentage, the scale for overhead gains is different w.r.t the one for accuracy loss.

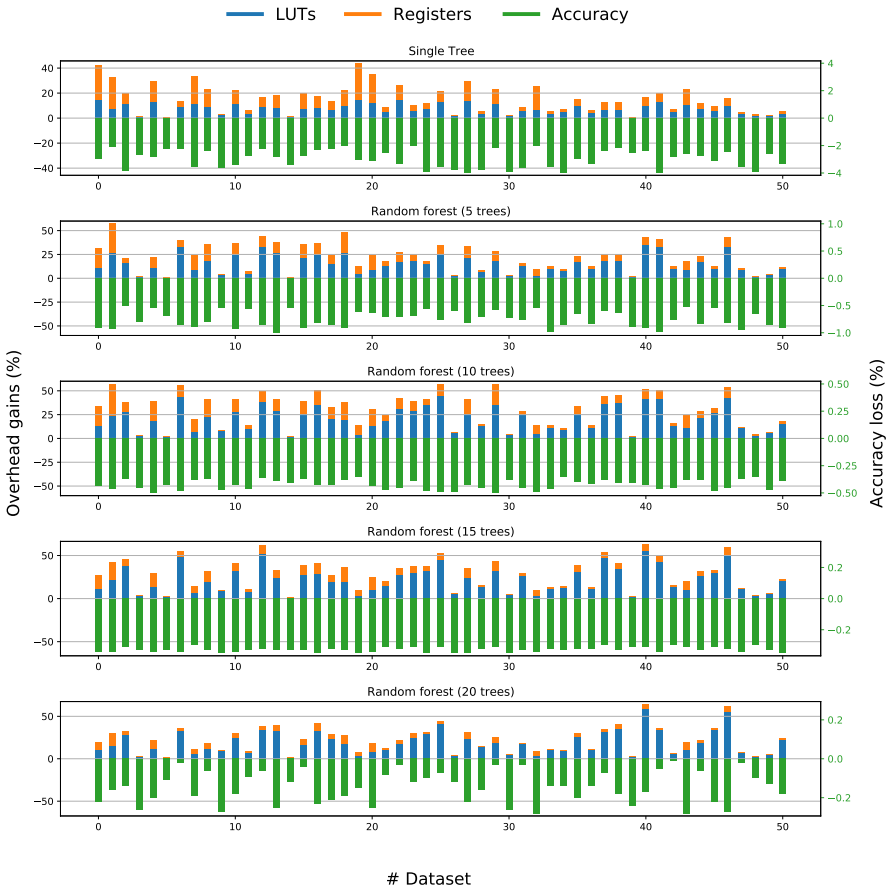



Figure 6.29: Amount of resource gain and accuracy loss for 50 different classification problems for maximum area overhead reduction approximate solutions. Please, kindly note that the scale on the left differs from the one on the right.

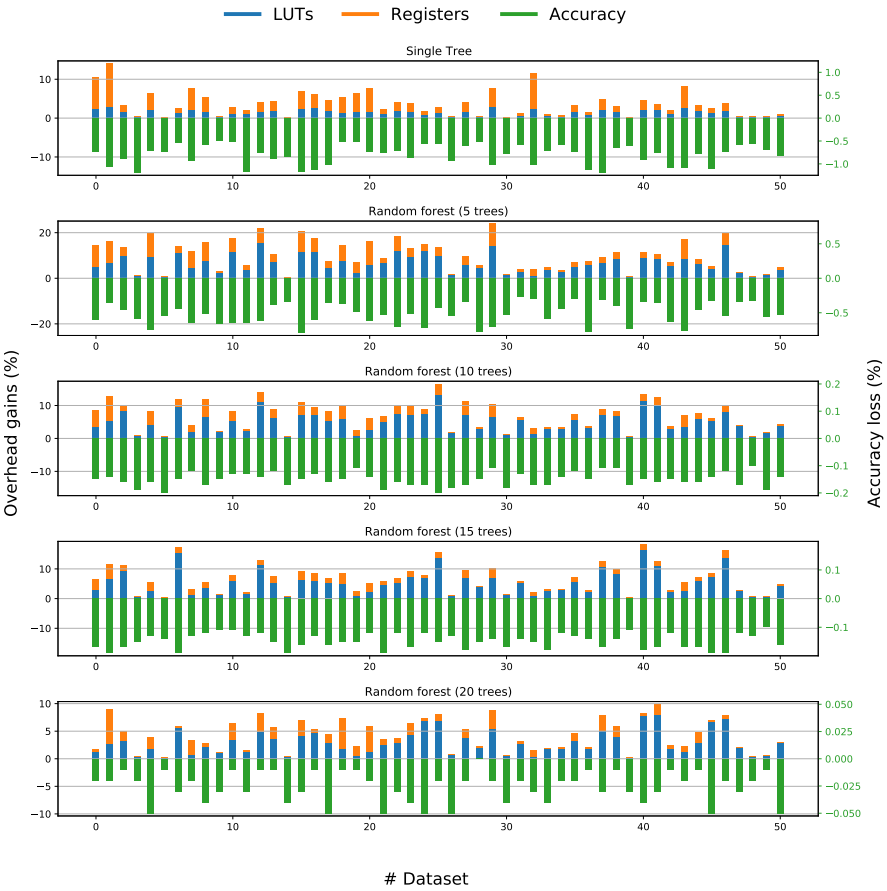



Figure 6.30: Amount of resource gain and accuracy loss for 50 different classification problems for minimum accuracy loss approximate solutions. Please, kindly note that the scale on the left differs from the one on the right.

For both graphs, we can state that accuracy loss decreases on the number of trees involved into the classification system. This observation confirms that random forest models are characterized by inherent resiliency property, and the greater the number of trees involved into models, the lower error introduced by approximation. Then, as for the reward, significant overhead reduction can be observed for random forest with 5, 10 and 15 trees, while the single DT model and 20 trees random forest exhibit lower

area reduction values. Indeed, single DT models cannot be conveniently approximate w.r.t. random forest models for the absence of a combiner that could mitigate the effect of introduced approximation. Nevertheless, contribution to area overhead of combiner circuits for random forest models with a significant number of trees makes the approximate computing technique less effective. As for solutions characterized by the minimum accuracy loss, we can see that even a small percentage of accuracy loss corresponds to a significant resource gain. As for synthesis of maximum accuracy loss solutions, we observe area reduction of more than 50% against about 0.2% of accuracy loss.

#### 6.2.6.4 Case-study #2: a SPAM detector

Since recognizing emails as SPAM or non-SPAM involves the classification of a large amount of information, a spam-detector case-study is used to evaluate the approach introduced in this paper. The data set used for this case-study is Spambase from the UCI Machine Learning Repository [86], which contains 4601 emails, 1813 of which are SPAM. This data set is freely available and makes use of 57 different features, expressed in the floating-point notation, to characterize elements that are part of the dataset. Each of the features specifies how often a word or a character appears in each element of the data set, i.e., in an email.

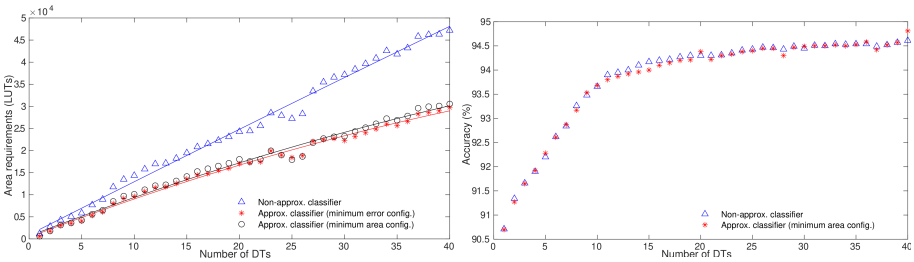
During the training phase, conducted using the KNIME [4] tool, 40 different random-forest classifiers with a number of DTs ranging from 1 to 40 are trained.

The AxC exploration phase found, for each of the 40 classifiers, a certain number of approximate configurations on the Pareto-frontier but for each of them only the configuration with minimum error and the one that requires less silicon area has been reported.

Figure 6.31a shows the area requirements in terms of LUTs, as the number of DTs used by the classifier increases. For all the measured quantities, an increasing trend, as the number of trees grows, is shown for area requirements. The growth, however, is clearly sublinear. In addition, it can be seen that the difference between requirements of the exact classifier and the approximate one increases as the number of trees grows. This is because even if the complexity of single DTs – i.e., the number of nodes of

which they consist of and the height of DTs themselves – decreases significantly as the number of trees used by the classifier increases, the total number of nodes increases, providing more approximation opportunities. This behavior can be observed also for the amount of FPGA slices and registers, and both when considering solutions providing minimum error and those requiring the minimum silicon area. Furthermore, it can be noted that the difference in terms of area requirements between the minimum error and the minimum area solutions always remains negligible.

Figure 6.31b compares the levels of classification accuracy, as the number of trees used by the classifier increases, provided by the precise version – without approximation – and by the approximate version that has minimum area requirements. It is evident, from the graph, that there is only a small difference in accuracy between the configurations. Moreover, it remains very small as the number of trees used for classification varies. On the other hand, the increase in the number DTs used in the classification process makes smaller contribution as the number of DTs grows. This asymptotic behavior can be seen in exact and approximate classifiers, and it is due to the fact that, by increasing the number of models, datasets involved for training turn out simpler and corresponding DTs get less branched, which leads to a saturation of the accuracy level provided by the classifier model.




(a) Required LUTs (b) Accuracy

Figure 6.31: Resource requirements and accuracy of approximate DT MCSs for Spambase [86]

#### 6.2.6.5 Comparison with previous works

In [23] a similar approach to the one presented in this paper has been adopted, but instead of exploring the solutions space with heuristics, the use of an exact algorithm,

namely B&B, was proposed. While, on the one hand, the use of an exact algorithm for the resolution of a MOP allows to reach a global optimal solution, on the other hand its use becomes prohibitive with large solutions spaces. Despite numerous improvements that the authors have made to the B&B implementation, such as pre-pruning of the tree and grouping features by information gain, they have managed to evaluate only a few classifiers, and for each of them only a few approximate configurations. This has greatly limited the quality of the solutions obtained. Table 6.15 shows classification error and hardware requirements in terms of LUTs for both approaches. As it can be observed, when compared to those obtained using NSGA-II, solutions provided by the B&B approach are worse. The difference in quality does not depend on the search algorithm itself, but on the amount of approximate configurations that have been taken into account during the space exploration phase.

Table 6.15: Comparison of results obtained from previous approaches

| | Approximate Minimum Error | | | | Approximate Minimum Area | | | |
|---|---|---|---|---|---|---|---|---|
| | Error | | LUTs | | Error | | LUTs | |
| DTs | B&B | GA | B&B | GA | B&B | GA | B&B | GA |
| 1 | 0 | 0 | 635 | 610 | 8,692E-4 | 1,0E-4 | 630 | 586 |
| 10 | 0 | 0 | 11853 | 10106 | 8,692E-4 | 3,0E-4 | 9646 | 9605 |
| 20 | 0 | 0 | 18091 | 18012 | 0 | 2,0E-4 | 18091 | 16996 |
| 30 | 0 | 0 | 23357 | 23243 | 4,346E-4 | 1,0E-4 | 23330 | 22243 |
| 40 | 0 | 0 | 33811 | 30544 | 8,692E-4 | 0,0E+0 | 30847 | 29747 |

# Chapter 7

# Conclusion

In this work, we discussed an application-independent, unified methodology able to automatically explore the impact of different approximation techniques on a given application, while resorting to the AxC design paradigm and MOP-based DSE. We discussed the steps the methodology breaks into while devoting particular relevance to all those that can be automated. In particular, we discussed how to effectively select parts of an application to be approximate and how to choose a well-suited approximation technique. Then, we discussed how approximate variants can be generated automatically, how to identify decision variable and suitable fitness-functions to define the MOP driving the DSE.

In order to evaluate the proposed metrology, we selected some significant and relevant applications in the scope of the AxC paradigm, including generic combinatorial logic, image-processing applications, and artificial intelligence applications.

For what pertains to generic logic, we propose local rewriting of AIG, reducing the number of nodes and resulting in lower hardware resources requirements, while resorting to MOP-based DSE to carefully introduce approximation. We evaluate our approach using different benchmarks, and, in order to measure actual gains, we perform FPGA synthesis of Pareto-optimal approximate configurations. Experimental results show our method allows performing a meaningful exploration of the design space to

find the best trade-offs in a reasonable time, thus resulting in approximate circuits exhibiting lower requirements and restrained error. Furthermore, our approach allows significant improvements over state-of-the-art works from the scientific literature.

Concerning image processing applications, two case studies are discussed. The first one aims at designing a Sobel edge-detector hardware accelerator, and it is of particular importance because the small size of the solution space allows the methodology to be compared against exhaustive exploration of the solution space. Experiments empirically proved results from our methodology are very close to those resulting from exhaustive simulation, while the amount of time needed to perform DSE is significantly lower. The second case study concerns the design of hardware-accelerators for the DCT, which is the most demanding step of the JPEG compression algorithm. We analyzed and modeled several algorithms from the literature to compute a fast and lightweight version of the DCT, and, for each algorithm, we applied approximation by substituting full-precise adders with several approximate ones from the literature having configurable approximation degree. In this way, we can obtain different approximate configurations of the algorithms, depending on the chosen approximate adders and their approximation degree. Approximate adders introduce inaccuracy in the computation, but also achieve gains in terms of area and power consumption. For each algorithm, we performed a DSE to find the non-dominated approximate designs in terms of trade-off between inaccuracy and gains. We modeled the DSE as a MOP and we used a GA to solve it. After the DSE, we synthesized the obtained designs by targeting both FPGA and ASIC. To do so, we implemented all the algorithms as re-configurable hardware designs. Finally, we evaluated the actual gains in terms of area and power consumption. Experimental results clearly showed that, with the proposed approach, it is possible to perform a meaningful DSE to find the best trade-offs between output accuracy and resource gains in a reasonable time. Finally, the comparison performed with previous work clearly showed the advantages of the proposed approach.

Last but not least, we applied our methodology to two of the most promising classification models in the machine-learning domain, namely DNNs and DT MCSs. Leveraging the AxC design paradigm, a very limited quantity of classification-accuracy is traded off for a reduction in the silicon area requirements and power consumption of hardware-implemented DT MCS and CNN. Concerning DNNs, we exploited our methodology to investigate the impact of several approximate techniques, including

precision-scaling, inexact-components and loop-perforation, on classification accuracy and hardware requirements. Furthermore, we investigated on the correlation between data-size and error resiliency. Experimental results prove the validity and efficiency of our methodology, even in applications in which the error has to be minimized as possible, providing savings up to 75% for silicon area and 50% for power consumption. Pertaining to DT MCSs, in order to prove the validity of the proposed approach several classifiers, with a number of trees ranging between 1 and 40, have been trained. Then, the optimal number of bits to be used to represent each of the features of the model is searched by means of NSGA-II. Among all Pareto-optimal hardware configurations, the one providing minimum classification error configuration and the one requiring the minimum amount of silicon area were taken into account for further consideration. Experimental results show a significant reduction in area requirements, for both the minimum error and minimum area configuration. Since the classification is very resistant to error, those configurations are very similar both in terms of area requirements and classification error.

# Bibliography


[1] 7nm Technology - Taiwan Semiconductor Manufacturing Company Limited. https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_7nm.

[2] Arithmetic Module Generator. https://www.ecsis.riec.tohoku.ac.jp/topics/amg/.

[3] Clang C Language Family Frontend for LLVM. https://clang.llvm.org/.

[4] Open for Innovation. https://www.knime.com/open-for-innovation-0.

[5] ORC Design and Implementation — LLVM 13 documentation. https://llvm.org/docs/ORCv2.html.

[6] SIPI Image Database. https://sipi.usc.edu/database/.

[7] 13 Sextillion & Counting: The Long & Winding Road to the Most Frequently Manufactured Human Artifact in History. https://computerhistory.org/blog/13-sextillion-counting-the-long-winding-road-to-the-most-frequently-manufactured-human-artifact-in-history/, April 2018.

[8] A. Aimar, H. Mostafa, E. Calabrese, A. Rios-Navarro, R. Tapiador-Morales, I. Lungu, M. B. Milde, F. Corradi, A. Linares-Barranco, S. Liu, and T. Delbruck. NullHop: A Flexible Convolutional Neural Network Accelerator Based on Sparse Representations of Feature Maps. *IEEE Transactions on Neural Networks and Learning Systems*, 30(3):644–656, March 2019.

[9] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *2016 ACM/IEEE 43rd Annual International Symposium*

*on Computer Architecture (ISCA)*, pages 1–13, Seoul, South Korea, June 2016. IEEE.

[10] Adrián Alcolea and Javier Resano. FPGA Accelerator for Gradient Boosting Decision Trees. *Electronics*, 10(3):314, 2021.

[11] Haider A.F. Almurib, T. Nandha Kumar, and Fabrizio Lombardi. Inexact designs for approximate low power addition by cell replacement. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 660–665, March 2016.

[12] Haider A.F. Almurib, T. Nandha Kumar, and Fabrizio Lombardi. Inexact designs for approximate low power addition by cell replacement. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 660–665, March 2016.

[13] Haider A.F. Almurib, Thulasiraman Nandha Kumar, and Fabrizio Lombardi. Approximate DCT Image Compression Using Inexact Computing. *IEEE Transactions on Computers*, 67(2):149–159, February 2018.

[14] Flora Amato, Mario Barbareschi, Valentina Casola, and Antonino Mazzeo. An fpga-based smart classifier for decision support systems. In *Intelligent Distributed Computing VII*, pages 289–299. Springer, 2014.

[15] Flora Amato, Mario Barbareschi, Valentina Casola, Antonino Mazzeo, and Sara Romano. Towards Automatic Generation of Hardware Classifiers. In Rocco Aversa, Joanna Kołodziej, Jun Zhang, Flora Amato, and Giancarlo Fortino, editors, *Algorithms and Architectures for Parallel Processing*, Lecture Notes in Computer Science, pages 125–132, Cham, 2013. Springer International Publishing.

[16] Yali Amit and Donald Geman. Shape Quantization and Recognition with Randomized Trees. *Neural Computation*, 9(7):1545–1588, October 1997.

[17] Mohammad Saeed Ansari, Vojtech Mrazek, Bruce F. Cockburn, Lukas Sekanina, Zdenek Vasicek, and Jie Han. Improving the Accuracy and Hardware Efficiency of Neural Networks Using Approximate Multipliers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(2):317–328, February 2020.

[18] Yukihiro Arai, Takeshi Agui, and Masayuki Nakajima. A fast DCT-SQ scheme for images. *IEICE TRANSACTIONS (1976-1990)*, 71(11):1095–1097, 1988.

[19] Sanghamitra Bandyopadhyay, Sriparna Saha, Ujjwal Maulik, and Kalyanmoy Deb. A Simulated Annealing-Based Multiobjective Optimization Algorithm: AMOSA. *IEEE Transactions on Evolutionary Computation*, 12(3):269–283, June 2008.

[20] Mario Barbareschi. Implementing Hardware Decision Tree Prediction: A Scalable Approach. In *2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 87–92, Crans-Montana, Switzerland, March 2016. IEEE.

[21] Mario Barbareschi, Salvatore Del Prete, Francesco Gargiulo, Antonino Mazzeo, and Carlo Sansone. Decision Tree-Based Multiple Classifier Systems: An FPGA Perspective. *International Workshop on Multiple Classifier Systems*, June 2015.

[22] Mario Barbareschi, Federico Iannucci, and Antonino Mazzeo. Automatic Design Space Exploration of Approximate Algorithms for Big Data Applications. In *2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 40–45, March 2016.

[23] Mario Barbareschi, Cristina Papa, and Carlo Sansone. Approximate Decision Tree-Based Multiple Classifier Systems. *International Conference on Optimization and Decision Science*, page 47, September 2017.

[24] Salvatore Barone, Marcello Traiola, Mario Barbareschi, and Alberto Bosio. Multi-Objective Application-Driven Approximate Design Method. *IEEE Access*, 9:86975–86993, 2021.

[25] F. M. Bayer and R. J. Cintra. DCT-like Transform for Image Compression Requires 14 Additions Only. *Electronics Letters*, 48(15):919, 2012.

[26] Saad Bouguezel, M. Omair Ahmad, and M. N. S. Swamy. Low-complexity 8$\times$ 8 transform for image compression. *Electronics Letters*, 44(21):1249–1250, 2008.

[27] Saad Bouguezel, M. Omair Ahmad, and M. N. S. Swamy. A fast 8×8 transform for image compression. In *2009 International Conference on Microelectronics - ICM*, pages 74–77, December 2009.

[28] Saad Bouguezel, M. Omair Ahmad, and M.N.S. Swamy. A low-complexity parametric transform for image compression. In *2011 IEEE International Symposium of Circuits and Systems (ISCAS)*, pages 2145–2148, May 2011.

[29] Robert Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pages 24–40. Springer, 2010.

[30] L. Breiman, J. H. Friedman, R. Olshen, and C. J. Stone. Classification and Regression Trees. *Routledge*, 1984.

[31] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, August 1996.

[32] Leo Breiman. Out-of-bag estimation. *Technical report, Statistics Department, University of California Berkeley, Berkeley CA*, page 13, 1996.

[33] Leo Breiman. Randomizing Outputs to Increase Prediction Accuracy. *Machine Learning*, page 14, 1998.

[34] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[35] Neil Burgess, Jelena Milanovic, Nigel Stephens, Konstantinos Monachopoulos, and David Mansell. Bfloat16 Processing for Neural Networks. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 88–91, June 2019.

[36] Sebastian Buschjäger and Katharina Morik. Decision Tree and Random Forest Implementations for Fast Filtering of Sensor Data. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(1):209–222, January 2018.

[37] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Muhammad Shafique, Guido Masera, and Maurizio Martina. An Updated Survey of Efficient Hardware Architectures for Accelerating Deep Convolutional Neural Networks. *Future Internet*, 12(7):113, July 2020.

[38] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pages 33–52, Indianapolis Indiana USA, October 2013. ACM.

[39] J. Castro-Godínez, H. Barrantes-García, M. Shafique, and J. Henkel. AxLS: A Framework for Approximate Logic Synthesis based on Netlist Transformations. *IEEE Transactions on Circuits and Systems II: Express Briefs*, pages 1–1, 2021.

[40] Lukas Cavigelli and Luca Benini. Origami: A 803-GOp/s/W Convolutional Network Accelerator. *IEEE Transactions on Circuits and Systems for Video Technology*, 27(11):2461–2475, November 2017.

[41] M. Češka, J. Matyaš, V. Mrazek, L. Sekanina, Z. Vasicek, and T. Vojnar. Approximating complex arithmetic circuits with formal error guarantees: 32-bit multipliers accomplished. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 416–423, November 2017.

[42] Arun Chandrasekharan, Mathias Soeken, Daniel Große, and Rolf Drechsler. Approximation-aware rewriting of AIGs for error tolerant applications. In *Proceedings of the 35th International Conference on Computer-Aided Design*, pages 1–8, Austin Texas, November 2016. ACM.

[43] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, January 2017.

[44] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

[45] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *arXiv:1410.0759 [cs]*, December 2014.

[46] V. K. Chippa, D. Mohapatra, K. Roy, S. T. Chakradhar, and A. Raghunathan. Scalable Effort Hardware Design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(9):2004–2016, September 2014.

[47] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference on - DAC '13*, page 1, Austin, Texas, 2013. ACM Press.

[48] Renato J. Cintra and Fábio M. Bayer. A DCT Approximation for Image Compression. *IEEE Signal Processing Letters*, 18(10):579–582, October 2011.

[49] Dan Cireşan, Ueli Meier, Jonathan Masci, and Jürgen Schmidhuber. Multi-column deep neural network for traffic sign classification. *Neural Networks*, 32:333–338, August 2012.

[50] Dan Ciresan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649, June 2012.

[51] Dan C. Cireşan, Alessandro Giusti, Luca M. Gambardella, and Jürgen Schmidhuber. Mitosis Detection in Breast Cancer Histology Images with Deep Neural Networks. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Kensaku Mori, Ichiro Sakuma, Yoshinobu Sato, Christian Barillot, and Nassir Navab, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2013*, volume 8150, pages 411–418. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[52] Dan C Ciresan, Ueli Meier, Jonathan Masci, Luca M Gambardella, and Jurgen Schmidhuber. Flexible, High Performance Convolutional Neural Networks for Image Classification. *Twenty-second international joint conference on artificial intelligence*, page 6, 2011.

[53] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Juergen Schmidhuber. Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition. *Neural Computation*, 22(12):3207–3220, December 2010.

[54] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv:1602.02830 [cs]*, March 2016.

[55] Jason Jinquan Dai, Yiheng Wang, Xin Qiu, Ding Ding, Yao Zhang, Yanzhang Wang, Xianyan Jia, Cherry Li Zhang, Yan Wan, Zhichao Li, Jiao Wang, Shengsheng Huang, Zhongyuan Wu, Yang Wang, Yuhao Yang, Bowen She, Dongjie

Shi, Qi Lu, Kai Huang, and Guoqiong Song. BigDL: A Distributed Deep Learning Framework for Big Data. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 50–60, Santa Cruz CA USA, November 2019. ACM.

[56] I. Das and J. E. Dennis. A closer look at drawbacks of minimizing weighted sums of objectives for Pareto set generation in multicriteria optimization problems. *Structural Optimization*, 14(1):63–69, August 1997.

[57] Ismael-Antonio Dávila-Rodríguez, Marco-Aurelio Nuño-Maganda, Yahir Hernández-Mier, and Said Polanco-Martagón. Decision-Tree Based Pixel Classification for Real-time Citrus Segmentation on FPGA. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, December 2019.

[58] dcadmin. Rebooting the IT Revolution: A Call to Action. https://www.semiconductors.org/resources/rebooting-the-it-revolution-a-call-to-action-2/.

[59] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. A Tutorial on Satisfiability Modulo Theories. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Werner Damm, and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590, pages 20–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[60] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.

[61] Kalyanmoy Deb, Karthik Sindhya, and Tatsuya Okabe. Self-adaptive simulated binary crossover for real-parameter optimization. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation - GECCO '07*, page 1187, London, England, 2007. ACM Press.

[62] Thomas G Dietterich. An Experimental Comparison of Three Methods for Constructing Ensembles of Decision Trees: Bagging, Boosting, and Randomization. *Machine learning*, 40(2):19, 1998.

[63] Stuart Dreyfus. The numerical solution of variational problems. *Journal of Mathematical Analysis and Applications*, 5(1):30–45, August 1962.

[64] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. ShiDianNao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 92–104, Portland Oregon, June 2015. ACM.

[65] Bradley Efron and Robert J. Tibshirani. *An Introduction to the Bootstrap*. Springer US, Boston, MA, 1993.

[66] Mohammed Elnawawy, Assim Sagahyroon, and Tamer Shanableh. FPGA-Based Network Traffic Classification Using Machine Learning. *IEEE Access*, 8:175637–175650, 2020.

[67] E. Feig and S. Winograd. On the multiplicative complexity of discrete cosine transforms. *IEEE Transactions on Information Theory*, 38(4):1387–1391, July 1992.

[68] Yoav Freund and Robert E. Schapire. Experiments with a New Boosting Algorithm, 1996.

[69] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, April 1980.

[70] Wulfram Gerstner and Werner M Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, Cambridge, U.K.; New York, 2002.

[71] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 383–397, Istanbul Turkey, March 2015. ACM.

[72] Vinayak Gokhale, Jonghoon Jin, Aysegul Dundar, Berin Martini, and Eugenio Culurciello. A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks. In

*2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 696–701, Columbus, OH, USA, June 2014. IEEE.

[73] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. SparTen: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 151–165, 2019.

[74] Vaibhav Gupta, Debabrata Mohapatra, Anand Raghunathan, and Kaushik Roy. Low-Power Digital Signal Processing Using Approximate Adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(1):124–137, January 2013.

[75] Vaibhav Gupta, Debabrata Mohapatra, Anand Raghunathan, and Kaushik Roy. Low-Power Digital Signal Processing Using Approximate Adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(1):124–137, January 2013.

[76] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[77] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both Weights and Connections for Efficient Neural Networks. *arXiv:1506.02626 [cs]*, October 2015.

[78] Muhammad Abdullah Hanif, Rachmad Vidya Wicaksana Putra, Muhammad Tanvir, Rehan Hafiz, Semeen Rehman, and Muhammad Shafique. MPNA: A Massively-Parallel Neural Array Accelerator with Dataflow Optimization for Convolutional Neural Networks. *arXiv:1810.12910 [cs]*, October 2018.

[79] Soheil Hashemi, Hokchhay Tann, and Sherief Reda. BLASYS: Approximate logic synthesis using boolean matrix factorization. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, San Francisco California, June 2018. ACM.

[80] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, Las Vegas, NV, USA, June 2016. IEEE.

[81] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher Fletcher. Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 674–687. IEEE, 2018.

[82] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, August 1998.

[83] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient Flow in Recurrent Nets: The Difficulty of Learning Long-Term Dependencies. *A Field Guide to Dynamical Recurrent Neural Networks*, March 2003.

[84] Sepp Hochreiter and Jurgen Schmidhuber. Bridging Long Time Lags by Weight Guessing and Long Short Term Memory". *Spatiotemporal models in biological and artificial systems*, page 8, 1996.

[85] Sepp Hochreiter and Jurgen Schmidhuber. Long Short-Term Memory. *Neural Computing*, 9(8):1735–1780, 1997.

[86] Mark Hopkins, Erik Reeber, George Forman, and Jaap Suermondt. Spambase Data Set. https://archive.ics.uci.edu/ml/datasets/spambase, 1999.

[87] Tong-Yu Hsieh, Kuen-Jong Lee, and M.A. Breuer. An error-oriented test methodology to improve yield with error-tolerance. In *24th IEEE VLSI Test Symposium*, pages 6 pp.–135, April 2006.

[88] D. H. Hubel and T. N. Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of Physiology*, 160(1):106–154, 1962.

[89] Taiga Ikeda, Kento Sakurada, Atsuyoshi Nakamura, Masato Motomura, and Shinya Takamaeda-Yamazaki. Hardware/Algorithm Co-optimization for Fully-Parallelized Compact Decision Tree Ensembles on FPGAs. In Fernando Rincón, Jesús Barba, Hayden K. H. So, Pedro Diniz, and Julián Caba, editors, *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, Lecture Notes in Computer Science, pages 345–357, Cham, 2020. Springer International Publishing.

[90] A. G. Ivakhnenko. Polynomial Theory of Complex Systems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-1(4):364–378, October 1971.

[91] Alexey Grigorevich Ivakhnenko. The group method of data of handling. *Soviet Automatic Control*, 13:43–55, 1968.

[92] Reinders J. Intel® AVX-512 Instructions. https://www.intel.com/content/www/us/en/develop/articles/intel-avx-512-instructions.html, 2021.

[93] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, Salt Lake City, UT, June 2018. IEEE.

[94] Honglan Jiang, Francisco Javier Hernandez Santiago, Hai Mo, Leibo Liu, and Jie Han. Approximate Arithmetic Circuits: A Survey, Characterization, and Recent Applications. *Proceedings of the IEEE*, pages 1–28, 2020.

[95] Weirong Jiang and Viktor K. Prasanna. Large-scale wire-speed packet classification on FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 219–228, 2009.

[96] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay

Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, Toronto ON Canada, June 2017. ACM.

[97] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, October 2016.

[98] Georgios Keramidas, Chrysa Kokkala, and Iakovos Stamoulis. Clumsy Value Cache: An Approximate Memoization Technique for Mobile GPU Fragment Shaders. *Workshop on approximate computing (WAPCO'15)*, page 6, 2015.

[99] Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. Zena: Zero-aware neural network accelerator. *IEEE Design & Test*, 35(1):39–46, 2017.

[100] Duckhwan Kim, Jaeha Kung, and Saibal Mukhopadhyay. A Power-Aware Digital Multilayer Perceptron Accelerator with On-Chip Training Based on Approximate Computing. *IEEE Transactions on Emerging Topics in Computing*, 5(2):164–178, April 2017.

[101] Scott Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics*, 34(5-6):975–986, March 1984.

[102] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[103] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 461–475, Williamsburg VA USA, March 2018. ACM.

[104] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4):541–551, December 1989.

[105] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[106] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[107] Yann LeCun, Corinna Cortes, and Chris Burges. MNIST Handwritten digit database. http://yann.lecun.com/exdb/mnist/, 1998.

[108] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H. Yoo. UNPU: An Energy-Efficient Deep Neural Network Accelerator With Fully Variable Weight Bit Precision. *IEEE Journal of Solid-State Circuits*, 54(1):173–185, January 2019.

[109] Jiajun Li, Shuhao Jiang, Shijun Gong, Jingya Wu, Junchao Yan, Guihai Yan, and Xiaowei Li. SqueezeFlow: A sparse CNN accelerator exploiting concise convolution rules. *IEEE Transactions on Computers*, 68(11):1663–1677, 2019.

[110] Arnaud Liefooghe, Matthieu Basseur, Laetitia Jourdan, and El-Ghazali Talbi. ParadisEO-MOEO: A Framework for Evolutionary Multi-objective Optimization. In Shigeru Obayashi, Kalyanmoy Deb, Carlo Poloni, Tomoyuki Hiroyasu, and Tadahiko Murata, editors, *Evolutionary Multi-Criterion Optimization*, volume 4403, pages 386–400. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[111] Xiang Lin, R.D. Shawn Blanton, and Donald E. Thomas. Random Forest Architectures on FPGA for Multiple Applications. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, GLSVLSI '17, pages 415–418, New York, NY, USA, May 2017. Association for Computing Machinery.

[112] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural Networks with Few Multiplications. *arXiv:1510.03009 [cs]*, February 2016.

[113] Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, 1976.

[114] CEA LIST. N2D2. CEA LIST, September 2021.

[115] Weiqiang Liu, Tian Cao, Peipei Yin, Yuying Zhu, Chenghua Wang, Earl E. Swartzlander, and Fabrizio Lombardi. Design and Analysis of Approximate Redundant Binary Multipliers. *IEEE Transactions on Computers*, 68(6):804–819, June 2019.

[116] C. Loeffler, A. Ligtenberg, and G.S. Moschytz. Practical fast 1-D DCT algorithms with 11 multiplications. In *International Conference on Acoustics, Speech, and Signal Processing,*, pages 988–991 vol.2, May 1989.

[117] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li. FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 553–564, February 2017.

[118] O. B. Lupanov. On a Method of Circuit Synthesis. *Journal of Symbolic Logic*, 35(4):593–594, 1970.

[119] Sana Mazahir, Osman Hasan, Rehan Hafiz, and Muhammad Shafique. Probabilistic Error Analysis of Approximate Recursive Multipliers. *IEEE Transactions on Computers*, 66(11):1982–1990, November 2017.

[120] Mitchell Melanie. An Introduction to Genetic Algorithms. *MIT Press*, page 162, 1998.

[121] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. Load Value Approximation. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 127–139, December 2014.

[122] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 532–535, July 2006.

[123] Alan Mishchenko, Satrajit Chatterjee, Roland Jiang, and Robert Brayton. FRAIGs: A Unifying Representation for Logic Synthesis and Verification. *ERL Technical Report*, page 7, 2015.

[124] Alan Mishchenko, Sungmin Cho, Satrajit Chatterjee, and Robert Brayton. Combinational and sequential mapping with priority cuts. In *2007 IEEE/ACM International Conference on Computer-Aided Design*, pages 354–361, November 2007.

[125] Sparsh Mittal. A Survey of Techniques for Approximate Computing. *ACM Computing Surveys*, 48(4):1–33, May 2016.

[126] V. Mrazek, M. A. Hanif, Z. Vasicek, L. Sekanina, and M. Shafique. autoAx: An Automatic Design Space Exploration and Circuit Building Methodology utilizing Libraries of Approximate Components. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2019.

[127] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina. EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 258–261, March 2017.

[128] V. Mrazek, L. Sekanina, and Z. Vasicek. Libraries of Approximate Circuits: Automated Design and Application in CNN Accelerators. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 10(4):406–418, December 2020.

[129] V. Mrazek, L. Sekanina, and Z. Vasicek. Using Libraries of Approximate Circuits in Design of Hardware Accelerators of Deep Neural Networks. In *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 243–247, August 2020.

[130] Vojtech Mrazek, Zdenek Vasicek, Lukas Sekanina, Muhammad Abdullah Hanif, and Muhammad Shafique. ALWANN: Automatic Layer-Wise Approximation of Deep Neural Network Accelerators without Retraining. *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, November 2019.

[131] Cody D. Murray and R. Ryan Williams. On the (Non) NP-hardness of Computing Circuit Complexity. *Theory of Computing*, 13(1):1–22, 2017.

[132] Hiroki Nakahara, Akira Jinguji, Simpei Sato, and Tsutomu Sasao. A Random Forest Using a Multi-valued Decision Diagram on an FPGA. In *2017 IEEE 47th International Symposium on Multiple-Valued Logic (ISMVL)*, pages 266–271, May 2017.

[133] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation*, 9(1):53–58, 2014.

[134] NVIDIA. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

[135] W. Osborne, J. Coutinho, W. Luk, and O. Mencer. Power-Aware and Branch-Aware Word-Length Optimization. In *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 129–138, April 2008.

[136] Muhsen Owaida, Amit Kulkarni, and Gustavo Alonso. Distributed Inference over Decision Tree Ensembles on Clusters of FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, 12(4):1–27, November 2019.

[137] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 45(2):27–40, 2017.

[138] Masoud Pashaeifar, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. A Theoretical Framework for Quality Estimation and Optimization of DSP Applications Using Low-Power Approximate Adders. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(1):327–340, January 2019.

[139] U S Potluri, A Madanayake, R J Cintra, F M Bayer, and N Rajapaksha. Multiplier-free DCT approximations for RF multi-beam digital aperture-array space imaging and directional sensing. *Measurement Science and Technology*, 23(11):114003, November 2012.

[140] Uma Sadhvi Potluri, Arjuna Madanayake, Renato J. Cintra, Fábio M. Bayer, Sunera Kulasekera, and Amila Edirisuriya. Improved 8-Point Approximate DCT for Image and Video Compression Requiring Only 14 Additions. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(6):1727–1740, June 2014.

[141] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 58–70, February 2020.

[142] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, March 1986.

[143] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[144] Arnab Raha, Swagath Venkataramani, Vijay Raghunathan, and Anand Raghunathan. Quality configurable reduce-and-rank for energy efficient approximate computing. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 665–670, March 2015.

[145] Abbas Rahimi, Andrea Marongiu, Rajesh K. Gupta, and Luca Benini. A Variability-Aware OpenMP Environment for Efficient Execution of Accuracy-Configurable Computation on Shared-FPU Processor Clusters, 2013.

[146] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan. ASLAN: Synthesis of approximate sequential circuits. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2014.

[147] Ashish Ranjan, Swagath Venkataramani, Shubham Jain, Younghoon Kim, Shankar Ganesh Ramasubramanian, Arnab Raha, Kaushik Roy, and Anand Raghunathan. Automatic Synthesis Techniques for Approximate Circuits. In Sherief Reda and Muhammad Shafique, editors, *Approximate Circuits: Methodologies and CAD*, pages 123–140. Springer International Publishing, Cham, 2019.

[148] Marc'Aurelio Ranzato, Fu Jie Huang, Y-Lan Boureau, and Yann LeCun. Unsupervised Learning of Invariant Feature Hierarchies with Applications to Object Recognition. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, June 2007.

[149] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. ASAC: Automatic sensitivity analysis for approximate computing. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems - LCTES '14*, pages 95–104, Edinburgh, United Kingdom, 2014. ACM Press.

[150] Sungju Ryu, Hyungjun Kim, Wooseok Yi, and Jae-Joon Kim. BitBlade: Area and Energy-Efficient Precision-Scalable Neural Network Accelerator with Bit-

wise Summation. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, Las Vegas NV USA, June 2019. ACM.

[151] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–50, Salt Lake City Utah USA, February 2014. ACM.

[152] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate data types for safe and general low-power computation. *ACM SIGPLAN Notices*, 46(5):11, 2011.

[153] Fareena Saqib, Aindrik Dutta, Jim Plusquellic, Philip Ortiz, and Marios S. Pattichis. Pipelined Decision Tree Classification Accelerator Implementation in FPGA (DT-CAIF). *IEEE Transactions on Computers*, 64(1):280–285, January 2015.

[154] I. Scarabottolo, G. Ansaloni, G. A. Constantinides, L. Pozzi, and S. Reda. Approximate Logic Synthesis: A Survey. *Proceedings of the IEEE*, 108(12):2195–2213, December 2020.

[155] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, January 2015.

[156] Lukas Sekanina, Zdenek Vasicek, and Vojtech Mrazek. Automated Search-Based Functional Approximation for Digital Circuits. In Sherief Reda and Muhammad Shafique, editors, *Approximate Circuits*, pages 175–203. Springer International Publishing, Cham, 2019.

[157] Sayeh Sharify, Alberto Delmas Lascorz, Kevin Siu, Patrick Judd, and Andreas Moshovos. Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, San Francisco California, June 2018. ACM.

[158] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775, June 2018.

[159] Zhao Shuang, Chen Shuhui, Yang Hui, Wang Fei, and Wei Zilin. RF-RISA: A novel flexible random forest accelerator based on FPGA. https://reader.elsevier.com/reader/sd/pii/S0743731521001477?token=0B1016BB744C0BD0FB89D99F1C307E761952586D95 west-1&originCreation=20210820065508, 2021.

[160] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering - SIGSOFT/FSE '11*, page 124, Szeged, Hungary, 2011. ACM Press.

[161] Mathias Soeken, Luca Gaetano Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. Exact Synthesis of Majority-Inverter Graphs and Its Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(11):1842–1855, November 2017.

[162] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[163] Da Tong, Yun Rock Qu, and Viktor K. Prasanna. Accelerating Decision Tree Based Traffic Classification on FPGA and Multicore Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 28(11):3046–3059, November 2017.

[164] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):273–286, June 2000.

[165] M. Traiola, A. Savino, M. Barbareschi, S. D. Carlo, and A. Bosio. Predicting the Impact of Functional Approximation: From Component- to Application-Level. In *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, pages 61–64, July 2018.

[166] Marcello Traiola, Alessandro Savino, and Stefano Di Carlo. Probabilistic estimation of the application-level impact of precision scaling in approximate computing applications. *Microelectronics Reliability*, 102:113309, November 2019.

[167] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei. Deep Convolutional Neural Network Architecture With Reconfigurable Computation Patterns. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(8):2220–2233, August 2017.

[168] Brian Van Essen, Chris Macaraeg, Maya Gokhale, and Ryan Prenger. Accelerating a Random Forest Classifier: Multi-Core, GP-GPU, or FPGA? In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 232–239, April 2012.

[169] Zdenek Vasicek. Formal Methods for Exact Analysis of Approximate Circuits. *IEEE Access*, 7:177309–177331, 2019.

[170] Zdenek Vasicek and Lukas Sekanina. Circuit Approximation Using Single- and Multi-objective Cartesian GP. *European Conference on Genetic Programming*, 9025:229, April 2015.

[171] Vassilis Vassiliadis, Konstantinos Parasyris, Charalambos Chalios, Christos D Antonopoulos, Spyros Lalis, Nikolaos Bellas, Hans Vandierendonck, and Dimitrios S Nikolopoulos. A Programming Model and Runtime System for Significance-Aware Energy-Efficient Computing. *ACM SIGPLAN Notices*, 50(8):2, 2015.

[172] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan. SALSA: Systematic logic synthesis of approximate circuits. In *DAC Design Automation Conference 2012*, pages 796–801, June 2012.

[173] Swagath Venkataramani, Anand Raghunathan, Jie Liu, and Mohammed Shoaib. Scalable-effort classifiers for energy-efficient machine learning. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, San Francisco California, June 2015. ACM.

[174] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. AxNN: Energy-efficient neuromorphic systems using approximate computing. In *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 27–32, August 2014.

[175] Z. Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, April 2004.

[176] J. Weng, N. Ahuja, and T.S. Huang. Cresceptron: A self-organizing neural network which grows adaptively. In *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, volume 1, pages 576–581 vol.1, June 1992.

[177] Paul J. Werbos. Applications of advances in nonlinear sensitivity analysis. In R. F. Drenick and F. Kozin, editors, *System Modeling and Optimization*, Lecture Notes in Control and Information Sciences, pages 762–770, Berlin, Heidelberg, 1982. Springer.

[178] Clifford Wolf and Johann Glaser. Yosys - A Free Verilog Synthesis Suite. page 6.

[179] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, January 2008.

[180] Q. Xu, T. Mytkowicz, and N. S. Kim. Approximate Computing: A Survey. *IEEE Design Test*, 33(1):8–22, February 2016.

[181] Saeyang Yang. *Logic Synthesis and Optimization Benchmarks User Guide: Version 3.0*. Citeseer, 1991.

[182] Zhixi Yang, Ajaypat Jain, Jinghang Liang, Jie Han, and Fabrizio Lombardi. Approximate XOR/XNOR-based adders for inexact computing. In *2013 13th IEEE International Conference on Nanotechnology (IEEE-NANO 2013)*, pages 690–693, August 2013.

[183] Zhixi Yang, Ajaypat Jain, Jinghang Liang, Jie Han, and Fabrizio Lombardi. Approximate XOR/XNOR-based adders for inexact computing. In *2013 13th IEEE International Conference on Nanotechnology (IEEE-NANO 2013)*, pages 690–693, August 2013.

[184] Amir Yazdanbakhsh, Divya Mahajan, Bradley Thwaites, Jongse Park, Anandhavel Nagendrakumar, Sindhuja Sethuraman, Kartik Ramkrishnan, Nishanthi Ravindran, Rudra Jariwala, Abbas Rahimi, Hadi Esmaeilzadeh, and Kia Bazargan. Axilog: Language support for approximate hardware design. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 812–817, March 2015.

[185] Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Onur Mutlu, and Todd C. Mowry. RFVP: Rollback-Free Value Prediction with Safe-to-Approximate Loads. *ACM Transactions on Architecture and Code Optimization*, 12(4):1–26, January 2016.

[186] T. Yeh, P. Faloutsos, M. Ercegovac, S. Patel, and G. Reinman. The Art of Deception: Adaptive Precision Reduction for Area Efficient Physics Acceleration. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 394–406, December 2007.

[187] Qian Zhang, Ting Wang, Ye Tian, Feng Yuan, and Qiang Xu. ApproxANN: An approximate computing framework for artificial neural network. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 701–706, March 2015.

[188] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-X: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[189] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: A comparative case study and the strength Pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, November 1999.