TUM School of Computation, Information and Technology
Technische Universität München

Approximate Computing for Motion Picture Camera Processing

Simon Conrady

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitz:
Prof. Dr.-Ing. Eckehard Steinbach

Prüfer der Dissertation:
1. apl. Prof. Dr.-Ing. Walter Stechele
2. Prof. Dr.-Ing. Ulf Schlichtmann

Die Dissertation wurde am 12.04.2022 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 01.10.2023 angenommen.

Abstract

Continuously growing demands for increased computational performance and improved power efficiency impose significant challenges for the design of digital systems. For digital motion picture cameras, these demands are driven by a steady trend towards capturing images at higher resolutions and frame rates as well as with more dynamic range while being battery-powered. As semiconductor process improvements face physical limitations and with the advent of the *dark silicon* phenomenon, engineers cannot solely rely on technological advancements anymore, which motivates alternative approaches in system design. To overcome these limitations, the idea of *Approximate Computing* describes a novel design paradigm which adds application quality as an orthogonal dimension into the traditional design trade-off between cost and performance. Relaxing the notion of correctness, it leverages the inherent error resilience of suitable applications by trading in accuracy to gain benefits in resource usage and/or performance. In recent years, numerous approximation methods have been proposed. While most of these methods are demonstrated in isolation, complex systems might benefit from a symbiotic combination of various approximations. However, the meaningful integration and parameterization of multiple approximations in such applications is a non-trivial problem as the corresponding design spaces grow exponentially and interactions between approximated system components need to be considered.

This dissertation proposes a framework which targets a purposeful combined integration and optimal parameterization of approximation methods in real-world applications, focusing on FPGA-based image processing systems. Several challenges at both the component and the application level are addressed by the contributions of this work.

Building upon a systematic survey of approximation techniques, promising methods are selected from the literature and implemented to form a library of approximate components. The implementations are optimized for the use in FPGA architectures and adapted to support a completely flexible scaling of both approximation strength and component size via direct parameterization, which is crucial for a meaningful combination with precision scaling in the same application. Furthermore, the proficiency of competing methods on the target platform is analyzed using an automated characterization of hardware and error properties, and machine learning-based models are trained to enable a fast fitness estimation.

On the application level, a data flow graph-based representation is proposed which allows for a systematic integration of approximations into target applications using node annotations. Furthermore, as the graph directly maps signal flow, it can be used to derive parameter dependencies and handle component interactions. To enable the necessary probing of many solutions during the design space exploration, the proposed resource models employ a divide-and-conquer approach to estimate the required area and power consumption. This omits the need for repeated synthesis, fitting and routing and hence accelerates the fitness estimation. In terms of quality assessment, the framework allows a flexible choice of any reference quality metric, allowing designers to obtain familiar application-specific values for their decisions instead of relying on abstract signal difference metrics.

A suitable optimization method based on the genetic algorithm is selected and integrated into the framework to enable an efficient design space exploration. The employed heuristic directly supports multi-objective optimization to handle the inherently conflicting goals of the quality-resource trade-off and uses a population-based global search to handle the navigation of complex design spaces. Three case studies are presented and used to evaluate the effectiveness of the proposed framework and to validate the suitability of the proposed models. The experimental results show that the framework is able to identify relevant solutions across a wide range of quality-resource trade-offs. A comparison between results from the full design space and after restrictions to single approximation types illustrates the benefits of a symbiotic combination of multiple methods.

Zusammenfassung

Kontinuierlich steigende Anforderungen an Rechengeschwindigkeit und Energieeffizienz stellen große Herausforderungen an den Entwurf digitaler Systeme. Im Bereich digitaler Bewegtbildkameras werden solche Anforderungen getrieben durch Trends zu höheren Auflösungen und Bildraten sowie zu einem größeren Dynamikumfang der aufgenommenen Bilder, worunter allerdings die Batterielaufzeit nicht leiden sollte. Da die Weiterentwicklung von Halbleiterprozesstechnik mittlerweile an pyhsikalische Grenzen stößt und der sogenannte *Dark Silicon*-Effekt an Bedeutung gewinnt, können Hardware-Entwickler sich nicht mehr alleine auf den technologischen Fortschritt verlassen. Diese Problemstellung motiviert alternative Ansätze im Entwurf digitaler Systeme. Die Idee von *Approximate Computing* stellt einen solchen Ansatz zur Überwindung der technischen Limitierungen dar. Sie beschreibt eine neuartige Systementwurfsphilosophie, welche die Anwendungsqualität als zusätzliche Dimension der traditionallen Balance zwischen Kosteneffizienz und Leistungsfähigkeit hinzufügt. Durch eine Aufweichung des Korrektheitsbegriffes wird damit die vorhandene Fehlertoleranz geeigneter Anwendungen ausgenutzt und Rechengenauigkeit eingetauscht, um Verbesserungen bei Energieeffizienz und/oder Rechenleistung zu erreichen. In den letzten Jahren wurde dazu eine Vielzahl an Approximationsmethoden verschiedener Art publiziert. Diese werden in der Regel einzeln demonstriert und evaluiert. In komplexen Systemen könnten jedoch durch synergetische Kombinationen mehrerer unterschiedlicher Methoden größere Verbesserungen erzielt werden. Die zielführende Integration und Parametrisierung mehrerer Approximationsmethoden in solche Anwendungen stellt jedoch ein nicht-triviales Problem dar, da der zugehörige Design Space exponentiell mit der Anzahl der verwendeten Methoden wächst und Wechselwirkungen zwischen approximierten Systemkomponenten berücksichtigt werden müssen.

Diese Dissertation befasst sich mit dieser Problemstellung und präsentiert ein Framework zur systematischen Integration und optimalen Parametrisierung von Approximationsmethoden in Anwendungen. Im speziellen fokussiert sie sich dabei auf FPGA-basierte Bildverarbeitungssysteme. Dabei löst sie verschiedene Herausforderungen, die sich sowohl auf Komponentenebene als auch auf Anwendungsebene ergeben.

Basierend auf einer systematischen Literaturübersicht vorgeschlagener Approximationsmethoden werden aussichtsreiche Methoden ausgewählt und implementiert, wodurch ein Baukasten mit approximierten Komponenten entsteht. Die Implementierungen sind für den Einsatz in FPGA-Architekturen optimiert und wurden so angepasst, dass eine flexible Skalierung sowohl der Komponentengröße als auch des Approximationsgrades durch direkte Parametrisierung möglich ist. Diese Eigenschaft ist wesentlich für eine sinnvolle Kombination der Komponenten mit Bitweitenskalierung innerhalb derselben Anwendung. Des Weiteren wird eine automatisierte Charakterisierung genutzt, um die Eigenschaften konkurrierender Methoden im Bezug auf die Zielplattform zu vergleichen. Um den Baukasten zu komplettieren werden mit Hilfe maschineller Lernmethoden Modelle erstellt, welche eine schnelle Abschätzung der Fitnesskriterien ermöglichen.

Auf der Anwendungsebene wird ein Signalflussgraph zur Systemmodellierung genutzt, welcher eine systematische Integration verschiedener Approximationsmethoden in die gewählte Anwen-

dung durch Annotation der jeweilgen Knoten ermöglicht. Da ein solcher Graph direkt den Signalfluss abbildet, können Abhängigkeiten und Interaktionen zwischen den Parametern unterschiedlicher Komponenten direkt abgeleitet und behandelt werden. Darauf aufbauend werden Ressourcenmodelle vorgestellt, welche den Ressourcenverbrauch und den Energiebedarf des Gesamtsystems basierend auf den Eigenschaften der einzelnen Systemkomponenten abschätzen, um eine effiziente Beurteilung vieler Lösungskandidaten zu ermöglichen. Der gewählte Ansatz umgeht den zeitaufwändigen Syntheseprozess, wodurch die Suche optimaler Konfigurationen erheblich beschleunigt wird. Für die Bewertung der Anwendungsqualität ermöglicht das Framework den Einsatz einer frei wählbaren Referenzmetrik, was dem Entwickler ermöglicht, mit vertrauten und etablierten, anwendungs-spezifischen Werten zu arbeiten, anstatt auf abstrakte Signalfehlermetriken zurückgreifen zu müssen.

Eine geeignete Optimierungsmethode basierend auf dem genetischen Algorithmus wird ausgewählt und in das Framework integriert, um eine effiziente Durchsuchung des Lösungsraumes zu ermöglichen. Die verwendete Heuristik verwendet ein Mehrzieloptimierungsverfahren, um die konkurrierenden Ziele hoher Bildqualität und niedriger Ressourcennutzung auszubalancieren und nutzt eine globale Suche, um durch komplexe Lösungsräume zu navigieren. Drei beispielhafte Fallstudien, welche reale Anwendungen abbilden, werden dazu genutzt, die Effektivität des Frameworks zu demonstrieren und die Tauglichkeit der verwendeten Modelle zu validieren. Die experimentellen Ergebnisse zeigen, dass die entwickelten Werkzeuge in der Lage sind, eine Vielzahl geeigneter Lösungen zu identifizieren, die einen weiten Bereich verschiedener Kompromisse zwischen Anwendungsqualität und Energiebedarf abdecken. Ein Vergleich mit Ergebnissen von zusätzlichen Experimenten, die auf einzelne Approximationsmethoden beschränkt sind, zeigt die Vorteile einer symbiotischen Kombination mehrerer Methoden auf.

Acknowledgment

Throughout the course of my research, I received a great deal of support, assistance and encouragement from many people without which this dissertation would not have been possible. Above all, I would like to express my deepest gratitude to my supervisor Prof. Walter Stechele for initiating this project and motivating me to start this endeavor. His guidance and scientific expertise continuously pushed my work forward while providing freedom to develop my own ideas, and his excellent feedback and academic advice improved the quality of my research. Furthermore, I would like to thank Prof. Ulf Schlichtmann for serving as my second examiner and Prof. Eckehard Steinbach for chairing the examination committee.

My appreciation goes to ARRI for allowing me to conduct my research as part of the R&D department, providing financial security and access to unparalleled expertise and domain knowledge. I would like to thank my mentor Dr. Tamara Seybold, whose engagement paved the way for starting this project and establishing my PhD position at ARRI. She continuously stimulated my scientific curiosity and supported me throughout this journey. I also would like to thank my manager Harald Brendel for his continuous support and all my colleagues from the Image Science group for welcoming me as part of the team. Special thanks go to Dominic Imm for sharing his FPGA development know-how and for our creative sessions at the whiteboard.

This work was conducted within the frame of a joint research project between ARRI, SmartRay GmbH and TUM. I would especially like to thank my project partners and fellow PhD students Manu Manuel and Arne Kreddig for their prolific and fruitful collaboration and continuous mutual support throughout the course of this project. All the countless discussions of creative ideas, implementation issues and publication details with you drove this project forward and helped me grow both professionally and personally. Furthermore, I would like to thank Dr. Anh Vu Doan for nudging us in the right direction concerning optimization strategies. Additional thanks go to all the students who contributed to this project, especially Mihai Babiac and Jiahui Xu. I would like to acknowledge the Bavarian Ministry of Economic Affairs, Regional Development and Energy for providing financial support (Grant No. IUK574) which made this project possible.

I want to thank my family and friends for their encouragement and practical support throughout the ups and downs that inevitably come within such a project. Special thanks to the residents of the WG in Schlierer Straße, who took me in during the pandemic and became valuable company in my day-to-day life. Thank you also to my parents and grandparents for their love and support throughout my entire life and for always believing in my abilities. Finally, I want to express my deepest heartfelt gratitude and appreciation to my wonderful wife Lena, for bringing joy and color into my days and for always having my back. I could not have reached this milestone without your consistent support and encouragement.

Contents

List of Acronyms

3D-LUT	3D lookup table
AA	affine arithmetic
ACA	accuracy-configurable adder
ALM	approximate logarithmic multiplier
ALS	approximate logic synthesis
AMA	approximate mirror adder
ASIC	application specific integrated circuit
AWG	Alexa Wide Gamut
AST	abstract syntax tree
AXA	approximate adder
BAM	broken-array multiplier
BBM	broken-Booth multiplier
BRAM	block RAM
CGP	cartesian genetic programming
CMOS	complimentary metal-oxide semiconductor
CORDIC	coordinate rotation digital computer
CPU	central processing unit
DCT	discrete cosine transform
DeMAS	design methodology for building approximate adders
DFG	data flow graph
DRAM	dynamic RAM
DRUM	dynamic range unbiased multiplier
DSE	design space exploration
DSP	digital signal processing
ED	error distance
EDP	energy-delay-product
EOTF	electro-optical transfer function
EPE	Early Power Estimator
ETA	error-tolerant adder

ETM	error-tolerant multiplier
EXDC	external don't care
FA	full adder
FAU	fast and error-optimized approximate adder units
FIR	finite impulse response
FPGA	field-programmable gate array
GA	genetic algorithm
GDA	gracefully degrading adder
GPU	graphics processing unit
HA	half adder
HBL	horizontal break level
HDL	hardware description language
HDR	high dynamic range
HDTV	high definition television
HLS	high-level synthesis
HOAANED	hardware-optimized approximate adder with near-normal error distribution
ILM	improved logarithmic multiplier
InXA	inexact adder
ISA	instruction set architecture
LM	logarithmic multiplier
LOA	lower-OR adder
LSA	lower-select adder
LSB	least significant bit
LUT	look-up table
MA	median adder
MaxED	maximum error distance
MaxRE	maximum relative error
MBM	minimally biased multiplier
MED	mean error distance
ML	machine learning
MLC	multi-level cell
MLP	multi-layer perceptron
MRE	mean relative error
MSB	most significant bit
MSE	mean squared error

NBias	normalized bias
NED	normalized error distance
NMaxED	normalized maximum error distance
NMED	normalized mean error distance
NPU	neural processing unit
NSGA-II	Nondominated Sorting Genetic Algorithm-II
OLOCA	optimized lower part constant-or adder
PSNR	peak signal-to-noise ratio
RAM	random access memory
RE	relative error
REALM	reduced-error approximate logarithmic multiplier
RF	random forest
RoBA	rounding-based multiplier
ROI	region of interest
ROI-NSGA	Region of Interest Nondominated Sorting Genetic Algorithm
RTL	register-transfer level
SNR	signal-to-noise ratio
SOA	set-one adder
SoC	system-on-chip
SRAM	static RAM
SSIM	structural similarity measure
SSM	static segment multiplier
STT-MRAM	spin transfer torque magnetic RAM
TIV	table of initial values
TO	table of offsets
UDM	underdesigned multiplier
VBL	vertical break level
VOS	voltage overscaling
XPE	Xilinx Power Estimator

Chapter 1

Introduction

Steadily increasing demands for more computational performance and reduced power consumption are outpacing technological improvements in computing systems. The conventional approach of shrinking transistor sizes, which has traditionally driven performance improvements across the semiconductor industry, is approaching physical limitations. With the breakdown of Dennard scaling [7], increasing the quantity of transistors per area leads to increased power densities. In turn, only parts of a chip can be running at full capacity at any given time – a phenomenon often described by the term *dark silicon* [8]. Hence, it is foreseeable that the growing demands of current and emerging applications cannot be met by advances in chip manufacturing alone. Mobile, battery-powered devices such as phones or cameras additionally have very strict power budgets to control heat dissipation and preserve battery life. Generally, designers of computing systems face a trade-off between area, power and computational performance. Since these are conflicting goals, the designer needs to carefully balance them to find a compromise that satisfies the needs of the target application. Due to the technological limitations discussed above, it is getting harder and harder to meet design targets within the traditional trade-off space, considering the continuous increase in computational demands of current and future applications.

In order to overcome these limitations, a novel design paradigm named *Approximate Computing* has emerged and gained a lot of traction in computing research in the last decade. The main idea of approximate computing is to extend the traditional design trade-off space with the dimension of application quality, as shown in Figure 1.1.

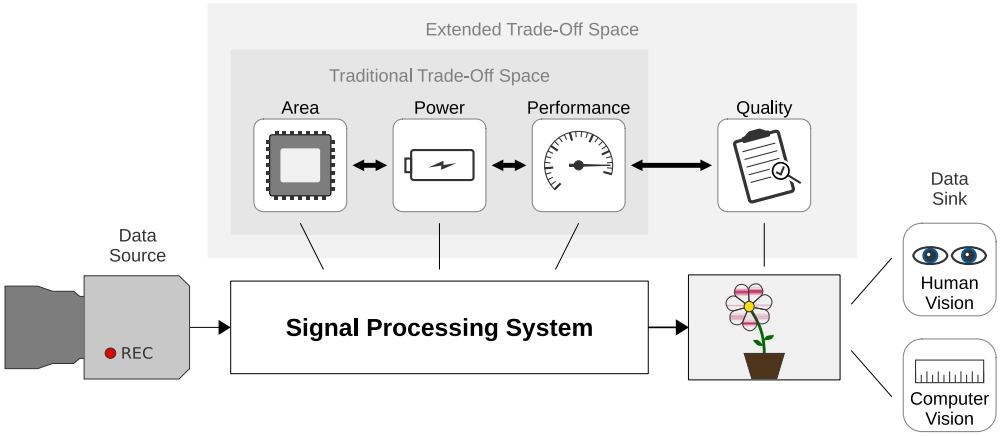


Figure 1.1: Approximate computing extends the traditional trade-off space of hardware design to include the quality dimension

Based on the assumption that certain applications are inherently resilient to errors, which means they can tolerate the loss of computational accuracy up to some degree, the imperative of creating 'exact' results is replaced with the intention of creating 'good enough' results. In the design process of a computational application, this concept is already widely used at the highest abstraction levels, i.e. application specification and algorithm design [9]. However, in the layers below, from the programming language, over the computer architecture down to the actual circuits, everything is expected to behave strictly correct and errors are treated as outliers that must not occur. Approximate computing breaks this rule by allowing imperfections to occur at every layer of the system stack. This enables the use of manifold new approximation methods, ranging from scaling the supply voltage below its threshold for safe behavior or adjusting the bitwidth of individual signals up to the replacement of code fragments or even entire applications with analog computation [10].

The notion of inherent application resilience to quality degradation is supported by several characteristics that allow for toleration of approximations, as shown in Figure 1.2. If one or more of these are present in a specific application, it qualifies for the use of approximate computing.

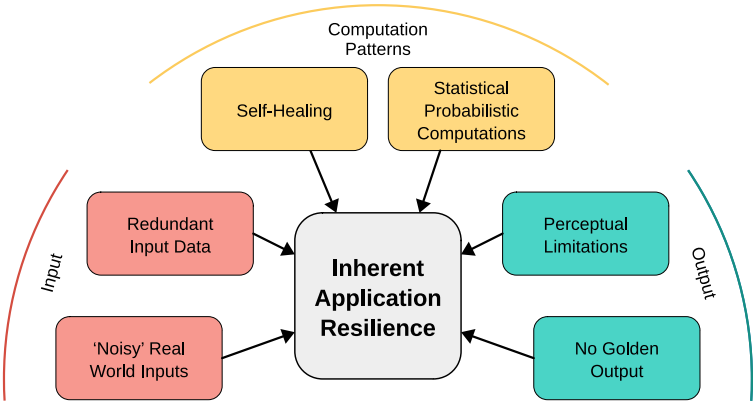


Figure 1.2: Factors contributing to inherent resilience of applications (adapted from [9] and [11])

First, some applications naturally deal with input data which is noisy and/or redundant. For example, systems that read and process data coming from a sensor such as a microphone or image sensor are specifically designed to handle the input signal noise, e.g. with measures to suppress it. The same holds for wireless communication systems that have to deal with input signals which degrade during the transmission. Since signal degradation caused by internal approximations is qualitatively similar to propagated input noise, the robustness of these applications can deal with both the same way. Furthermore, such applications often process redundant input data, either implicitly (i.e. by the nature of the captured data) or explicitly (e.g. to enable error correction in weak communication channels), which may offer additional robustness.

Moreover, some applications inherently exhibit self-healing properties by the nature of their structure, e.g. iterative refinement algorithms, which may automatically attenuate or remove errors introduced with approximations. This is especially true in applications that apply statistical computations across a large amount of input data, for example in Neural Networks, where many values are aggregated and errors may cancel each other out.

Lastly, there are several possible limitations regarding the interpretation of the output data produced by the application. There could be no single golden output and hence a range of answers may be equally acceptable, e.g. in search or recommendation engines. On the other hand, if there

is a golden answer, in some cases it might be unlikely to be found even by a perfect system, which is true for most machine learning systems. In this case, end users are also likely to accept a good-enough result. For example, computer vision systems in which camera images are used for 3D measurements, object detection and tracking or similar purposes may tolerate some quality loss as long as the general functionality is guaranteed. Finally, limitations in human perception may give way to the notion that sub-perfect results may be sufficient since they could not be distinguished from perfect results by the end user. Typically, multimedia processing systems fall into this category.

1.1 Target Application Scope

Digital motion picture camera systems fit well into the scope of approximate computing as current trends push the necessity of alternative solutions to overcome technological limitations. In this field, the demands manifest in continuously growing requirements for the spatial as well as temporal resolution together with the dynamic range of the recorded content. The dynamic range defines the ratio between the lightest and the darkest useful luminance signal captured in each pixel, which translates to the number of bits needed for each pixel. Furthermore, the spatial resolution determines the number of pixels in each image and the temporal resolution dictates the number of images captured and processed per second. All of these factors culminate in a considerably high data throughput that needs to be processed in the camera. Additionally, many motion picture production scenarios require powering the cameras from batteries because a wired connection to the electrical grid is not always practical, driving a demand for low power consumption that stands diametrically opposed to the push for higher performance.

On the other hand, image processing applications exhibit several of the factors described above contributing to their resilience to quality degradation. The captured data is inherently affected by photon shot noise and dark current noise [12]. Additionally, non-linear internal image processing steps, e.g. scene-referred encoding of colors or luminance tone-mapping may increase or decrease signal deviations depending on the actual pixel value [13]. Hence, if applied carefully, approximations may benefits from self-healing properties of the image processing pipeline. Finally, images recorded by motion picture cameras for entertainment purposes are ultimately being consumed by humans with perceptual limitations.

This work concentrates on the application of approximate computing within embedded low-level image processing systems implemented using field-programmable gate arrays (FPGAs) as found in professional digital motion picture cameras. Such systems comprise a variety of different operations across multiple processing stages and are commonly implemented as *stream processing* pipelines to achieve the necessary data throughput required for high resolutions and frame rates [14]. This means that all operations are “unrolled” and placed in parallel on the chip into a pipelined design that accepts a new pixel in every cycle and returns it after processing with a delay of multiple cycles. The data flow graph (DFG) of the application is an abstracted representation of the structure of the corresponding circuit. We target the integration and combination of different parameterizable single-purpose approximation methods into such systems, focusing on deterministic approximations employed directly on-chip within the data flow of the application.

Technical Notes The selection and implementation of approximate components as well as entire applications carried out later in this work targets current commercial FPGA architectures of

different vendors in general and does not depend on vendor-specific technologies and design primitives to enable usability across devices. However, throughout this work, we use a specific FPGA device from the Intel Arria 10 product family [15], namely the `10AS066N3F40E2SG` device, which is found on the evaluation board in the Intel Arria 10 SoC Development Kit [16], as target device and experimental platform. Without loss of generality, we set the operating frequency to 266.66 MHz for all experiments and fix the board temperature to 50°C for power analysis.

1.2 Research Problems

This work is mainly motivated by the fact that even though a substantially large body of research on approximate computing methods already exists, there is still a crucial gap between the predominantly theoretical proposal of various approximation methods on the one hand and their meaningful practical integration and combination within real-world FPGA-based image processing systems on the other. In detail, this dissertation strives to address and answer the following questions:

How to effectively approximate within FPGA designs? The vast majority of published approximation methods targeting embedded hardware computing systems are designed for application specific integrated circuit (ASIC) implementations and exploit the degrees of freedom offered in such completely custom designs. While FPGAs are able to implement custom logic at a very low level and in a fine-grained manner, their architecture imposes constraints on the practical realization of circuits. FPGA implementations use specific building blocks such as logic modules containing look-up tables (LUTs) and registers for combinatorial logic, digital signal processing (DSP) slices for arithmetic operations and embedded block RAM (BRAM) for fast storage. Approximation methods should use these resources effectively, but ASIC-focused designs do not necessarily translate into beneficial FPGA implementations. Therefore, a selection of suitable methods and their meaningful adaption towards the general FPGA architecture are needed. Additionally, many published approximate components, especially arithmetic units, are only available in specific sizes, which hinders their flexible scaling and hence a meaningful combination with fine-grain precision scaling. To overcome this limitation, the FPGA-based implementation should be generically parameterizable to adapt to arbitrary sizes. Finally, since the quality-resource trade-off performance translation of methods transferred from ASIC to FPGA designs is uncertain, the resulting implementations have to be characterized systematically and the results compared for competing methods that target the same operation.

How to model the quality-resource trade-off efficiently and accurately in systems that combine multiple approximation types? Secondly, most of the proposed approximation methods have been demonstrated and characterized well in isolation, but optimal exploitation of the quality-resource trade-off might necessitate a controlled combination of multiple methods. However, most state-of-the-art methods for the parameterization of approximated hardware systems are restricted to methods of a single type.

In real-world applications, there are structural cross-dependencies between different system components when the bitwidth of intermediate signals connecting them is affected by approximations, either directly using precision scaling or indirectly as a side effect of an approximated operation. These dependencies need to be taken into account in the system parameterization as well as when modeling the resource consumption.

Secondly, to enable meaningful choices regarding the acceptable quality loss, the quality model should provide data that is relevant for the application and interpretable for the designer. Related work on approximate computing often measures quality in the form of basic signal fidelity metrics such as the error rate, the relative error or the hamming distance. While these metrics are able to measure and quantify errors, their implications for real-world applications are not always clear, especially when human perception is involved in the judgment. Instead, application-specific metrics would allow for designers to use familiar means of quality assessment which have been tried and tested within their application domain [17].

Generally, to enable an efficient exploration of complex design spaces, the fitness models used to quantify the quality-resource trade-off must be fast and reliable. In terms of speed, the employed models should avoid time-consuming synthesis and gate-level simulation.

How to optimize approximation parameters in complex design spaces? The error propagation and interactions between components mentioned above prevent an isolated optimization of parameters for individual system components. Consequentially, the design space grows exponentially with the number of employed approximations and their parameter ranges, necessitating the use of an automated DSE. Since resource savings and application quality are competing objectives, there is no single optimal solution. Instead, a number of Pareto-optimal solutions exist, from which the desired trade-off can be chosen. Therefore, an appropriate multi-objective optimization algorithm must be selected to facilitate the search and exploitation of the quality-resource trade-off offered by the target application.

1.3 Contributions and Thesis Outline

Addressing the problems formulated above, this thesis proposes a framework for the systematic integration and optimal parameterization of approximations in FPGA-based signal processing applications, focusing on low-level image processing pipelines. The framework has matured over the past years and was demonstrated at different stages in several publications [1–4]. This thesis presents an updated version that contains additional improvements over the previous publications.

The design flow of the proposed framework follows three main phases, as shown in Figure 1.3. Structurally, this thesis first mirrors these phases, covering each of them in a dedicated chapter as indicated in the figure. Following the methodological description, three practical case studies covering common image color processing tasks are introduced and evaluated to demonstrate the proposed approach, before a final chapter concludes the thesis and outlines directions for future research. The contributions and contents of each phase/chapter are summarized below.

Phase I: Component Level (Chapter 2)

The first phase is application-independent and deals with the formation of a *library of approximate components* which can be employed across many different applications. Correspondingly, it covers the path between the initial publication of approximation methods and their flexible and reusable implementation for FPGA-based systems. The existing body of literature proposing a wide range of approximation methods for hardware-based computing systems is reviewed thoroughly and in a structured manner. Several methods suitable for the target application field and applicable for FPGA designs are selected and implemented for the library. The selected techniques span different

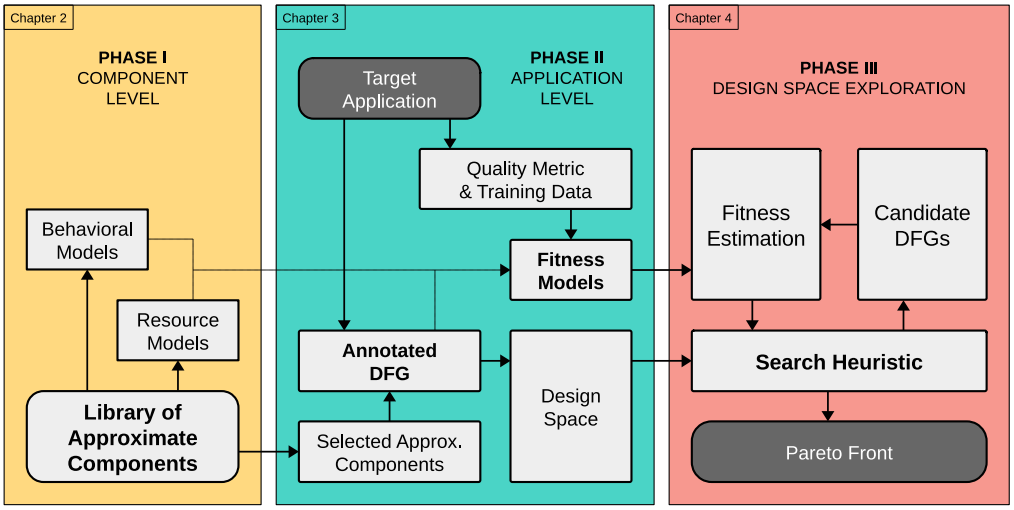


Figure 1.3: Design flow phases of the proposed framework

categories of approximation methods, namely *precision scaling*, *arithmetic units* and *table-based methods*. Generally, the approximation methods directly translate to approximate components except for precision scaling, which affects the width of signals between system components. The implementations of methods originally proposed for ASIC designs are adapted to the characteristics of FPGA architectures where needed. In contrast to related work, our implementations provide flexible parameterization in terms of approximation strength **and** component size (in terms of I/O signal width) so that they can be scaled flexibly and hence employed in combination with precision scaling in applications.

Furthermore, an automated procedure for the characterization of approximate components w.r.t. a specific target FPGA platform is used to provide further information for the refined selection of components for a target application. The characterization data is used to comparatively analyze the trade-offs associated to the selected approximate arithmetic units when used in FPGAs, providing insights into how well different methods adapt to the FPGA architecture. As a second purpose, the characterization data is used to create resource models for components whose consumption cannot be derived from their configuration in a straightforward manner. The library formed after these steps contains a parameterizable hardware description, a *behavioral model* for accelerated bit-exact simulation and a *resource model* for each approximate component.

Phase II: Application Level (Chapter 3)

The second design phase deals with the integration of approximations into a target application and models the quality-resource trade-off related to any parameterization on the application level. To that end, we propose a structured representation of the application in form of an *annotated DFG* where the designer can mark individual nodes for approximation and define the range of their parameters. Hence, the design space formed by the annotated DFG is given by the set of parameters across all nodes and its complexity scales with their number and ranges. To handle structural interaction between connected components, we present a systematic approach to capture the propagation of parameter dependencies, from which the correct parameterization order is derived. The

proposed representation and dependency handling of the application allows the simultaneous deployment and parameterization of multiple approximation types in contrast to related work, which is typically either limited to a specific type, e.g. arithmetic approximations, or employs different approximation types in a sequence of distinct steps.

In order to enable an efficient evaluation of probed candidate solutions during the DSE, we propose fast models to estimate the application quality and resource consumption based on the annotated DFG representation. A divide-and-conquer strategy is proposed to model the usage of FPGA resource units using only the characteristic consumption of individual components provided in the library without the need for time-consuming synthesis of the application. The power consumption is then directly derived from the estimated number of FPGA resources. To ensure a meaningful assessment of application quality, the proposed framework enables the designer to choose their preferred reference quality metric. Hence, the quality estimation provides values that are relevant for the application and easily understood and interpreted by the designer. Additionally, we discuss the choice of relevant training data with respect to the target application scope and propose two training set types tailored towards different quality statistics. The quality model in the framework uses the annotated DFG together with the behavioral models of the individual components to provide a fast, bit-exact simulation of the parameterized application that can be assessed using any provided metric function and automatically considers error propagation effects.

Phase III: Design Space Exploration (Chapter 4)

Combining multiple approximations in the same application results in a complex design space which necessitates an effective and automated DSE procedure. Generally, a search heuristic generates *candidate DFGs* which represent specific parameterizations of the annotated DFG, estimates their fitness using the application-level models established in phase II and uses the results in a feedback loop to generate better candidates. This process repeats until eventually Pareto-optimal or near Pareto-optimal solutions have been found. The proposed framework employs a genetic algorithm (GA)-based heuristic to guide the search which directly integrates multi-objective optimization and is able to handle complex design spaces efficiently. Necessary adaptions for the genetic operations to respect the parameter dependencies are described in detail.

The optimization methodology selected for the proposed framework was developed mainly by Manuel as part of a joint collaborative research project. It was demonstrated in a journal publication [5] and is not claimed as a contribution of this work. However, its integration into the proposed framework helps to navigate the arising complexity of the tackled design spaces.

Case Studies (Chapter 5)

Chapter 5 contains the experimental demonstration and evaluation of the proposed framework in real-world applications. Three case studies of image color processing pipelines are presented and optimized using the framework, resulting in Pareto fronts that provide solutions across a wide variety of different quality-resource trade-offs to choose from. The results are also used to demonstrate the benefits of combining multiple approximation methods. Since the optimization estimates the fitness of individual solutions based on fast and simple models, the resource consumption is validated against post-synthesis results and the quality estimation is compared to the values obtained from a real-world image set.

Chapter 2

Approximate Components

The main idea of approximate computing is to allow a certain amount of imprecision in the computations of an application in order to achieve benefits in resource consumption and/or performance. Research in this field has gained a lot of traction in recent years, generating an immensely large number of publications which provide different ideas and methods for introducing approximations into computing systems. The research body consequently spans a wide range of abstraction levels, target platforms, and application fields and selecting suitable methods for a specific design project can be overwhelming. While there are methods that directly target FPGA implementations, they only represent a very small portion of the available literature. Nevertheless, many other methods, e.g. approximations defined on the algorithmic level or circuit approximations targeting implementation on ASIC, can potentially be leveraged for FPGA-based systems. However, their adaptability to specific FPGA architectures and the resulting benefits are often not immediately clear. This chapter deals with the gap between the publication of individual approximation methods and the ability to employ them in real-world FPGA-based applications. The general structure of this path is depicted in Figure 2.1, which also mirrors the structure of this chapter.

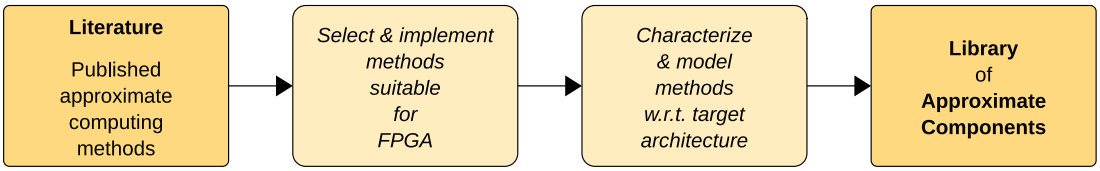


Figure 2.1: Overview of steps necessary to make published approximation methods usable in FPGA-based signal processing systems

In the first section of this chapter, a review of existing literature on approximation methods generally suitable for hardware-based systems is given. To make use of the methods contained in the literature, they must first be filtered based on their potential to achieve benefits when used in FPGAs, which requires knowledge of the FPGA architecture as well as the target application domain. For this work, several promising methods with relevance for FPGA-based signal processing systems are selected and described in detail in the second section. Furthermore, to understand the specific quality-resource trade-off offered from any of these methods, they must be characterized in terms of resource usage (in terms of area and power) as well as error behavior. This also allows comparing multiple competing methods targeting the same operation, e.g. different approximate multiplier techniques. Therefore, the third section describes a methodology to automatically perform this characterization w.r.t. the target FPGA architecture and analyzes the results. To allow estimating the resource consumption of all implemented components, Section 2.4 introduces and analyzes

machine learning (ML) approaches to create models for the cases in which the consumption cannot be derived analytically. Lastly, a library of approximate components and building blocks which can be reused across different applications is formed, containing parameterizable hardware implementations together with behavioral models and resource models. Section 2.5 provides details on the library implementation and its contents.

2.1 State of the Art in Approximation Methods

Since the interest in the field of approximate computing has grown immensely over the last decade or two, so has the number of publications proposing a wide variety of approximation methods. A multitude of surveys exist that provide overviews of the ongoing research [10, 18–20]. Such overviews generally cover the entire body of related literature. In contrast, the overview presented here focuses on the approximation methods directly targeted at or generally applicably to hardware-based computing systems that implement the application as a custom circuit, which means that specific software-level methods, approximation-enhanced central processing unit (CPU) architectures and graphics processing unit (GPU)-specific methods are not covered here.

To structure the remaining literature, the methods are classified according to the abstraction level. We group the methods into the main categories of *device* level, *circuit* level and *algorithm* level, breaking the latter down further into *elementary functions* and *control flow* sub-categories. Furthermore, approximation methods for targeting *memory* are listed separately. A tabular overview of methods, sorted by classification, is given in Table 2.1.

Table 2.1: Categorized overview of approximation methods targeted at hardware systems

Category	Method	FPGA	Parameterizable	Deterministic	References
Device	Voltage Overscaling	◪[1]	■	□	[21–24]
Circuit	Precision Scaling	■	■	■	[25–28]
	Circuit Pruning	■	□	■	[29–33]
	Approximate Logic Synthesis	■	□	■	[34–43]
	Approximate Arithmetic Units	■	■	■	[44–76]
Algorithm (Elementary Functions)	Polynomial Approximation	■	■	■	[77–79]
	Incremental Refinement	■	■	■	[80–82]
	Table-Based Methods	■	■	■	[83–87]
Algorithm (Control Flow)	Fuzzy Memoization	■	■	■	[88, 89]
	Neural Acceleration	◪[2]	□	■	[90–92]
Memory	Memory Voltage Overscaling	◪[1]	■	□	[93–96]
	Refresh Rate Scaling	◪[3]	■	□	[97–99]
	Approximate Memory Operation	◪[3]	■	□	[100, 101]
	Memory Access Scaling	◪[3]	■	■	[102]

[1] Typically, only global voltage scaling is feasible on current commercial FPGAs, hindering controlled adaption [96]
[2] Needs a dedicated neural accelerator to yield optimal benefits
[3] Can be done in off-chip memory, not in the FPGA chip itself

The table also provides assessments of the suitability of the methods for FPGA designs, and their scalability, meaning the ability to easily parameterize both the signal width (for functional blocks) and the degree of approximation, and whether they behave deterministically or not. Additionally, key references are listed for each method. The following subsections provide descriptions and details for all methods, grouped by the abstraction level.

2.1.1 Device Level

Approximations on the device or transistor level encompass techniques for *voltage overscaling (VOS)* or *frequency overscaling*. These two methods can be seen as different perspectives on the same principle, namely the acceptance of timing errors and their optimization, with the former emphasizing energy reduction and the latter focusing on performance improvements. Without loss of generality, we focus on explicit VOS techniques in this overview, which are described below. Note that in this section we describe VOS techniques employed on the computational data-path. Similar methods used to reduce energy consumption of memories are listed separately with the *Memory Level* techniques in Section 2.1.5.

2.1.1.1 Voltage Overscaling (VOS)

The idea of VOS is to reduce the supply voltage of a circuit below the nominal level in order to reduce the dynamic power consumption, which typically dominates the total power consumption and can be expressed as

$$P_{\text{dyn}} \approx \alpha \cdot C_{\text{L}} \cdot f_{\text{clk}} \cdot V_{\text{dd}}^2 \tag{2.1}$$

with the activity factor α, the load capacitance of the circuit C_{L}, the clock frequency f_{clk} and the supply voltage V_{dd} [103]. Consequentially, a reduction in supply voltage leads to a quadratic reduction in dynamic power consumption. However, reducing the voltage also leads to an increase of circuit delay which in turn can lead to erroneous outputs due to timing violations on the critical paths. Existing techniques for VOS therefore aim at minimizing the probability of violations to control the output quality.

Kahng et al. propose a method for *slack redistribution* to allow for graceful degradation of the output under VOS [21]. They argue that traditional circuit design methodologies lead to a wall of slack, i.e. many long combinatorial paths with low slack margins, as illustrated in Figure 2.2. With VOS, the zero-slack margin shifts which would result in a large number of failing paths for traditional designs and consequentially lead to poor application quality. In contrast, their methodology adapts circuit gate sizes in order to reshape the slack distribution to have a more gradual slope so that timing violations happen less frequently under VOS.

The work proposed by Mohapatra et al. focuses on the design of scalable *meta-functions*, i.e. computational kernels that are commonly used across many applications [22]. First, they propose to cut the carry propagation between adder segments in an accumulator depending on the supply voltage in order to prevent the propagation of glitches resulting from timing violations. To compensate potential errors introduced by the split in the carry chain, a multi-cycle error compensation circuit is used. Secondly, they introduce a delay budgeting technique which splits a combinatorial circuit into two distinct blocks which are characterized individually to extract their error behavior with reduced delay budgets. If the first block behaves more gracefully than the second, a transparent latch is placed between the blocks which is used to freeze the first block's output after a portion

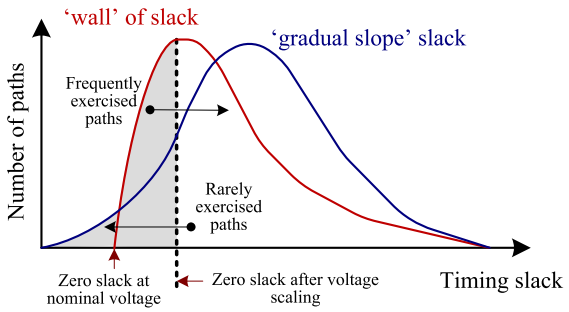


Figure 2.2: Slack distribution under traditional design approaches that lead to a 'wall' of slack (red) and target 'gradual slope' slack distribution (blue) for graceful degradation under VOS (Source: [21], © 2010 IEEE)

of the clock period, giving the second block more time for its computations. This effectively redistributes the delay budget within a single clock period between the two logic blocks to improve the output quality.

To accelerate design flows that utilize VOS, Zervakis et al. proposed *VOSsim*, a technique for behavioral simulation and power estimation of circuits under sub-threshold voltage scaling [23]. Their technique uses re-characterization of the employed technology library at border voltages in combination with the Synopsys composite current source timing model to estimate circuit path delays for any intermediate voltage value. Furthermore, they extend the behavioral simulation by characterizing the behavior of flip-flops in the case of timing violations to replace unknown outputs (reported as `X`) with an estimated value (`0` or `1`) depending on the flip-flop type, the transition type and the violation time. In [24], the authors use VOSsim to combine VOS with other approximation techniques for individual nodes in the DFG of the target application.

The methods for VOS-based approximation described above target ASIC designs and assume that the supply voltage can be scaled in a fine-grained manner or at least differently for individual sub-circuits or voltage islands. However, in current commercial FPGA platforms, the supply voltage that feeds the logic fabric together with the hard DSP units is controlled globally for the entire device, which restricts the usefulness of these methods [96].

2.1.2 Circuit Level

The next higher level of approximation methods refers to all low-level techniques that directly alter the circuit implemented for the targeted application in the selected hardware device. In our classification, we include *Precision Scaling*, *Approximate Arithmetic Units*, *Circuit Pruning* and *Approximate Logic Synthesis*, which are introduced and discussed in the following subsections.

2.1.2.1 Precision Scaling

Arguably the most traditional way of adapting the accuracy in computations is to scale the precision of the data being processed. In CPU software, a binary floating-point representation is often used to approximately describe real numbers, which are encoded using the sign s, the exponent e and the mantissa m as

$$x_{\text{real}} \approx x_{\text{float}} = (-1)^s \cdot 2^e \cdot m. \tag{2.2}$$

The exponent dynamically controls the position of the fractional point of the number with respect to its magnitude. Hence, this representation yields the same relative precision for small and large numbers. The length of the mantissa m used in the encoding defines the amount of relative precision offered in the representation. Designers can typically choose between different formats which are supported in the hardware architectures of common CPUs and GPUs, namely double precision (64 bits, with 52 bits for m), single precision (32 bits, with 23 bits for m) and half precision (16 bits, with 10 bits for m). However, the dynamic scaling of the exponent causes significant overhead in arithmetic operations with floating-point numbers and makes the required hardware more complex. Furthermore, being constrained to a set of fixed formats may waste resources if intermediate accuracy would be sufficient for the application.

In contrast, hardware-based signal processing systems that implement custom circuits as ASIC or on FPGAs often use a fixed-point data representation. Here, the position of the fractional point is fixed and a real number is approximated as

$$x_{\text{real}} \approx x_{\text{fixed}} = \text{round}(x_{\text{real}} \cdot 2^n) \cdot 2^{-n} = X \cdot 2^{-n}, \tag{2.3}$$

where X is the binary integer carried by the signal and n is the number of fractional bits associated with it. This format allows all numbers to be treated as plain integers for all arithmetic operations, removing the overhead that comes with a floating-point representation. However, the rounding operation in Equation 2.3 introduces quantization noise [104] which inversely scales with the number of fractional bits (the quantization step is $\Delta x = 2^{-n}$). Hence, the precision of the signal depends on its fractional bitwidth.

In a custom circuit implemented using fixed-point arithmetic, the bitwidth of every signal in the system can be tuned to the needed precision, which can be referred to as *fine-grain precision scaling*. In complex systems, however, the problem of assigning optimal bitwidths to individual signals is a non-trivial task. Addressing this problem, *Minibit* [25] uses affine arithmetic (AA) [105] to model the precision of internal signals in an application. First, as initial coarse-grain optimization, the optimum uniform fractional bitwidth (i.e. assigning the same precision to every signal) that respects the target output precision is calculated analytically using AA. In a second, fine-grain optimization step, an adaptive simulated annealing algorithm is used to tune the bitwidth of individual signals in order to reduce the cost of the circuit further. Another technique, *LenghtFinder* [26] also starts with a coarse-grain analysis that determines the minimal uniform bit-length. From there, it uses a set of heuristic methods to perform fine-grain precision optimization in reduced time. Optionally, an additional optimization stage can be used to improve the results using a genetic algorithm, which is shown to outperform the simulated annealing approach.

In contrast to the approaches described above, where the precision is optimized statically and the circuit is implemented with fixed signal bitwidths, other works aim at dynamically scaling the precision during runtime. Park et al. describe a method for scaling the precision in the calculation of discrete cosine transform (DCT) coefficients used for data compression [27]. First, they use a heuristic algorithm to assign different bitwidths to the calculation of the coefficients within three fixed quality-resource trade-off levels. To achieve dynamic reconfigurability, they add a pair of additional control transistors to specific gates implemented in complimentary metal-oxide semiconductor (CMOS) logic so that selected functional blocks can be switched off. Hence, the user can switch between the predefined trade-off levels at runtime while the control consumes only little area and delay overhead.

Furthermore, Lee and Gerstlauer proposed an approach that specifically targets applications defined by mean squared error (MSE)-type error metrics that may accept noisy input signals [28]. The main idea is that precision scaling can be applied to keep the quality level constant at a desired level for all input conditions (defined by the set of input signal-to-noise ratios (SNRs)) that would lead to higher quality levels than required. They use an additive noise model that represents input noise and quantization noise by their variance and uses propagation through adders and multipliers to calculate the output variance. On the other hand, power consumption is derived from the sum of unit hardware blocks used by the individual operations in the system, depending on the bitwidth at the operation's output. To find an optimal set of bitwidths that meet a desired quality level while minimizing the consumed power, they apply adaptive simulated annealing as in [25]. This approach is used to compute optimal bitwidths for any given combination of input SNRs and target output SNR. Then, either one of the solutions can be statically implemented or a table with different solutions can be stored in the system so that it can react to changes in the input SNRs. In the latter case, the bitwidths of the selected signals are dynamically controlled via clock gating.

2.1.2.2 Circuit Pruning

Several methods have been proposed aiming at simplifying arbitrary combinatorial circuits. They start from a complete circuit and prune cells to minimize the resource consumption while meeting a specified quality target. Most of these methods work iteratively, removing circuit elements in an order that promises to yield the best resource-quality trade-off.

To that end, *probabilistic pruning* [29] was proposed, a heuristic method that uses simulations to identify the circuit elements with the lowest probability of being active and removes them until a defined quality threshold is violated. In order to consider the error magnitude together with the error rate, all circuit elements can be assigned a significance value that reflects their impact on the output. The pruning can then be done after ranking the significance-activity products. Schlachter et al. extended this methodology by providing automatization tools and integrating it into a standard design flow [30]. In a similar approach, May and Stechele use an FPGA-based accelerated fault emulation system [106] to identify registers that can be removed from a circuit without violating a given output error constraint [31]. All logic that only feeds into a pruned register can be removed as well. Using the fault emulation system also allows identifying tolerable error probabilities at registers that cannot be completely removed which allows combining the pruning with VOS methods.

A different method called *optimal slope ranking* [32] estimates the effect of pruning each cell in the circuit in terms of power, delay and error. With that, an approximate efficiency value is calculated as the ratio between the energy-delay-product (EDP) and the increase in error. Then, the pruning candidates are ranked according to the approximate efficiency and the cell with the highest value is pruned. This process repeats until the error threshold is reached.

In contrast to the previous techniques which iteratively remove parts of the circuit, *Circuit Carving* searches globally for the maximum portion or *cut* of a circuit that can be replaced with constant values so that a given maximum error is respected [33]. To find the best cut, the technique explores a binary search tree representing all possible cuts and uses heuristics to reduce the number of branches to visit before finding the optimal solution.

2.1.2.3 Approximate Logic Synthesis (ALS)

Another type of methods directed at the approximation of arbitrary circuits consists of techniques to directly synthesize approximate circuits, also denoted as approximate logic synthesis (ALS) methods. An early approach in this area by Shin and Gupta proposes a heuristic method to find optimal minterm complements (i.e. modifying specific outputs in the truth table so that a minterm is simplified) which lead to a reduction in circuit area under a given error rate constraint when synthesized [34].

Another technique called *SALSA* [35] reformulates the approximate synthesis problem into a standard synthesis problem that can be solved by standard synthesis tools. It uses a quality function that calculates a validity bit from the accurate and approximate outputs for any input, reporting whether the quality bound is met or not. For each output of the approximate circuit, *approximation don't cares* are found, which are the set of input values to which the output of the quality function remains insensitive. These are then specified as external don't cares (EXDCs) and hence standard don't care-based synthesis can be used to synthesize the approximate circuit. Similarly, Miao et al. propose to use EXDCs to perform multi-level approximate logic synthesis [36]. Using Boolean relations to model the allowed error magnitude behavior, they start from an overly relaxed set of EXDCs and iteratively find an optimal EXDC set that respects the error magnitude constraint. Additionally, their method supports error frequency constraints by successively recovering correct outputs until the respective constraint is satisfied.

SASIMI [37] is a different approach from the same research group that proposed *SALSA*, aiming at generating quality-configurable circuits. Their approach selects pairs of signals that take on the same values with a high probability, and replaces the target signal by the substitute signal. In the approximate mode, logic elements feeding only into the target signal can be removed and others may be downsized due to relaxed timing constraints. Additionally, there is a quality configurable mode which keeps both signals but downsizes the gates generating the target signal to save power. If accurate operation is chosen then the system monitors for errors and gates the clock for one additional cycle to recover occurring errors. In approximate operation, errors are ignored and the system operates in a single cycle.

Employing evolutionary optimization, Vasicek et al. propose to use a cartesian genetic programming (CGP) [107] algorithm to generate approximate circuits. In CGP, the composition of a circuit is encoded as a chromosome that can be evolved using the GA. Different variants were studied, aiming at optimizing the area given a fixed error constraint [38] or vice versa [39]. A detailed summary of their work is given in [40]. They also demonstrated that benefits obtained by their method applied at the gate level are preserved when the circuits are synthesized for FPGAs [41].

A different methodology named *BLASYS* [42] uses boolean matrix factorization to split an accurate multi-output circuit into a compressor circuit which computes a number of feature signals and a decompressor circuit which recovers the outputs. The degree of approximation can be controlled by selecting the number of features, with less features yielding more approximation. In order to support error magnitude minimization, they modified existing factorization algorithms to support the notion of bit significance. Furthermore, to reduce computational complexity when approximating large circuits, heuristic methods are used to decompose the circuit and to select the order in which the resulting sub-circuits are approximated.

Extending the combinatorial ALS problem to sequential circuits, *ASLAN* [43] provides a modified quality function that takes the state of the circuit into account to ensure that the approximate circuit eventually reaches completion. To generate the approximate circuit, combinatorial blocks

are heuristically selected for approximation and quality-energy trade-off graphs are generated for all candidates using existing combinatorial approximation techniques. A gradient descent algorithm then chooses the candidate with the best quality-energy trade-off. This repeats until the quality function signals that the quality constraint is violated and/or the approximate circuit fails to reach completion.

2.1.2.4 Approximate Adders

Arithmetic units are among the most important building blocks of signal processing systems. Especially adders and multipliers are prominently used operations and hence numerous methods for their approximation have been proposed by the research community. This section will therefore review and discuss different options of approximate adders, while the subsequent section focuses on approximate multipliers.

In this overview, we classify the proposed approximate adders into four different categories. The first contains designs for *approximate full adder (FA)* cells. In the second category, the addition is split into an accurate and an approximate segment, cutting the carry chain and simplifying the logic in the approximate segment. These units are denoted as *two-segment carry-split adders*. The next group contains the *multi-segment adders*, which contain multiple carry splits to reduce the delay even further. Finally, libraries of *evolved adders* have been proposed that contain circuits generated by an evolutionary ALS procedure. Table 2.2 provides an overview over selected methods across these categories which are described below.

Approximate Full Adder Cells The first sub-category of approximate adders builds upon modified designs of FA cells, which are the fundamental building blocks of many conventional adder designs. By allowing errors in the truth table, simplified designs of the respective circuits are possible. Approximate adders can then be built by replacing some of the accurate FA cells with approximate ones.

Gupta et al. propose four different *approximate mirror adder (AMA)* designs by removing transistors from a conventional mirror adder design to allow various quality-resource trade-offs to be achieved [44]. The approximate designs consume different amounts of area and the adders built from them have different critical path lengths, which allows a reduction in supply voltage for further savings. In a similar approach, but starting from efficient XOR/XNOR-based FA designs, Yang et al. propose three different approximations, simply denoted as *approximate adder (AXA)*, reducing the transistor count even further at comparable output deviations [45]. Lastly, Almurib et al. propose three additional approximate FA designs, denoted as *inexact adder (InXA)* [46]. Their cells are designed from scratch and slightly reduce the number of transistors and the number of erroneous outputs compared to the previous designs.

However, it should be noted that while adders built from approximate FA cells can deliver beneficial quality-resource trade-offs in ASICs, they are not advantageous in FPGA designs which do not allow modifications at the transistor-level granularity and can implement accurate FAs in a single LUT, which is the smallest available architectural building block in FPGAs.

Specifically targeting FPGA-designs, Prabakaran proposed a *design methodology for building approximate adders (DeMAS)* [47] which contains eight different designs for one-bit and two-bit adder building blocks from which larger multipliers can be built. While outperforming FPGA implementations of ASIC-based FAs, their building blocks are specifically designed with the ar-

Table 2.2: Overview of approximate adders

Abbreviation	Full Name	Reference
	Approximate Full Adders	
AMA	Approximate Mirror Adder	[44]
AXA	Approximate Adder	[45]
InXA	InExact Adder	[46]
DeMAS	Design Methodology for Building Approximate Adders for FPGA-based systems	[47]
	Two-Segment Carry-Split Adders	
ETA-I	Error-Tolerant Adder-I	[48]
LSA	Lower-Select Adder	[44]
MA	Median Adder	[49]
LOA	Lower-Or Adder	[50]
—	Sloppy Adder	[51]
OLOCA	Optimized Lower Part Constant-Or Adder	[52]
HOAANED	Hardware-Optimized Approximate Adder with Near-Normal Error Distribution	[53]
FAU	Fast and Error-Optimized Approximate Adder Units	[54]
	Multi-Segment Adders	
ETA-II	Error-Tolerant Adder-II	[55]
ACA	Accuracy-Configurable Adder	[56]
GDA	Gracefully Degrading Adder	[57]
GeAr	Generic Accuracy-Configurable Adder	[58]
	Evolved Adders	
EvoApprox8b	Library of evolved 8-bit adders and multipliers	[59]
EvoApproxLib	Extended library of evolved adders and multipliers of various sizes	[60]

chitecture of Xilinx 7-series FPGAs in mind and are not usable across the FPGA architectures of different vendors.

Two-Segment Carry-Split Adders While the approximate adders described in the previous section replace some of accurate FAs with simplified designs, they do not change the underlying architecture of the adder itself. Specifically, they preserve the carry propagation chain. Contrasting these methods, there are many approaches that are based on the same fundamental idea: splitting the adder into multiple segments and cutting the carry chain at the split points. Splitting the carry chain lowers the critical delay of the adder and at the same time reduces power consumption as less switching activity is caused by glitches in the carry chain [48].

Most proposed designs split the adder into exactly two parts, and the upper part is calculated accurately using a conventional adder while the lower part is approximated in different ways. The approximate part also provides an estimate of the carry-in for the accurate part. Figure 2.3 depicts the common architecture of these designs, given the overall bitwidth of the addition n and the split-

ting point k. The inputs are denoted as A and B while the sum is denoted as S, and specific bit positions are indicated in the subscript of these variables.

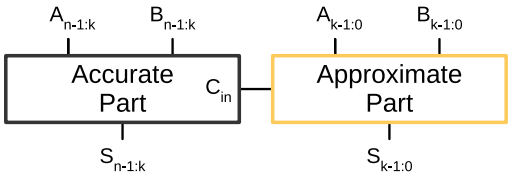


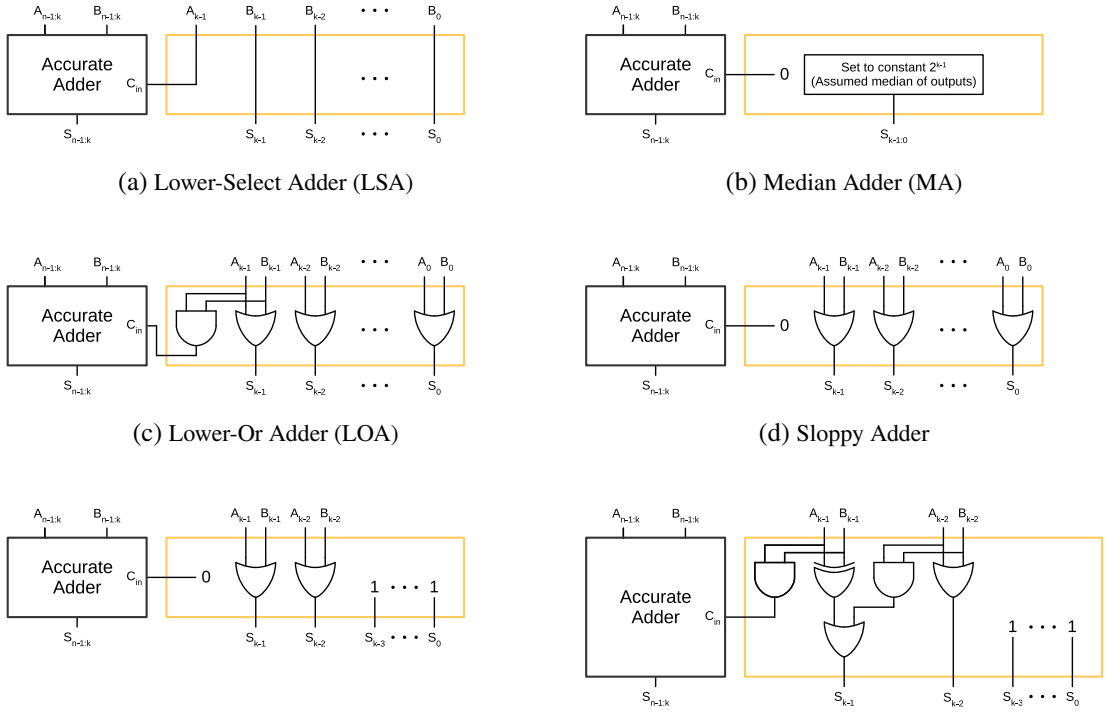
Figure 2.3: Common architecture of carry-split adders

Zhu et al. proposed the *error-tolerant adder (ETA)-I* design, which reverts the direction of operation in the lower part [48]. More specifically, the mechanism of generating the approximate lower part sum proceeds from the most significant bit (MSB) to the least significant bit (LSB). The sum at each position is set to `1` if either one of the respective inputs is `1` and to `0` if both inputs are zero. However, if both inputs are `1`, then the output of the current bit and the all sum bits to the right are set to `1`, which reduces the overall error of the result. In the actual implementation, the individual sum bits are calculated by means of modified XOR gates that feature an additional control input which can force the output to `1`. Hence, an additional control block is needed to generate individual control signals for each bit position in the lower part, which adds considerable logic overhead.

Subsequent works target simpler lower-part approximations, yielding a large number of very similar designs, as shown in Figure 2.4. Together with their AMA designs mentioned above, Gupta et al. also presented a fifth approximation, in which the full adders in the approximate part are completely removed [44]. Hence, the sum of the lower part is approximated by one of the inputs, while the carry-in for the upper part is connected to the lower-part-MSB of the other input, as shown in Figure 2.4a. This effectively replaces the lower part logic with wires. Since the choice which input to forward to the output can be parameterized to adapt to the structure of the application, we will denote this adder as *lower-select adder (LSA)*. The structure of this may lead to further simplifications in the application, as only the MSB of the discarded input needs to be calculated in preceding parts of the circuit.

Similarly, the *median adder (MA)* [49] also completely removes the logic in the lower part, replacing it by the constant binary representation of its expected median output, as shown in Figure 2.4b. Assuming a uniform distribution for the input portions fed to the lower part, the expected median output is given as $2^k - 1$. This results in all output bits being constantly set to `1` while the carry-in to the upper part is set to zero. If the distribution of inputs is known more exactly, the expected median can be calculated directly and a different constant lower-part value can be set. This adder is even simpler than the LSA as it directly connects the lower-part outputs to V_{DD} or ground and may lead to simplifications in both input paths.

In the *lower-OR adder (LOA)* [50], each bit of the lower-part sum is approximated by the OR function of the respective input bits. Additionally, the carry-in to the upper part is computed as the AND function of the lower-part MSBs of both inputs, effectively reducing the worst case error by roughly 50%. Very similar to the LOA, Albicocco et al. proposed the *sloppy adder* [51], which generally operates in the same way but does not predict a carry-in to the accurate part (see Figure 2.4d). While this reduces the cost by one gate, it increases both the average and the worst-case errors (for the same value of k).



(a) Lower-Select Adder (LSA)

(b) Median Adder (MA)

(c) Lower-Or Adder (LOA)

(d) Sloppy Adder

(e) Optimized Lower Part Constant-Or Adder (OLOCA)

(f) Hardware-Optimized Approximate Adder with Near-Normal Error Distribution (HOAANED)

Figure 2.4: Various carry-split adder architectures

Starting from the LOA structure, the *optimized lower part constant-or adder (OLOCA)* [52] replaces the OR-gate producing the lower-part MSB sum bit with an XOR gate. Because this effectively means using a half adder (HA) in this position, it is the same as decreasing the split point k by one and feeding a `0` as carry into the upper part, as in the sloppy adder described above. However, their structure additionally replaces the OR gates in the lowest portion of the approximate part by constant `1`s, similar to the approach of the MA. They find that the optimal split between OR gates and constant `1`s is after two OR gates, counting from the left. The resulting architecture is depicted in Figure 2.4e. Recently, the *hardware-optimized approximate adder with near-normal error distribution (HOAANED)* [53] was introduced, featuring an architecture similar to the OLOCA, but replacing the two OR gates at the MSBs of the approximate part with more sophisticated logic (see Figure 2.4f). The top-most approximate bit position now behaves like a half-adder whose sum output is feed into an OR gate together with an estimated carry produced by an AND gate from the position to its right. This leads to a well-behaved error distribution with an average error close to zero, which means that the output deviation is not strongly biased.

Specifically targeting FPGA designs, Echavarria et al. propose fast and error-optimized approximate adder units (FAU) [54]. In their design, the addition is also split into a lower and an upper part, but both parts perform accurate addition, making the broken carry chain the only potential source of errors in the system. To reduce the error, a carry is predicted using a parameterizable amount of MSB inputs to the lower part which are shared into the upper part. Furthermore, in cases where the lower part produces a carry which would not be caught by the prediction, all sum

bits in the lower part are forced to `1`. While this solution increases the speed of the adder, it also introduces additional control overhead for the error reduction schemes.

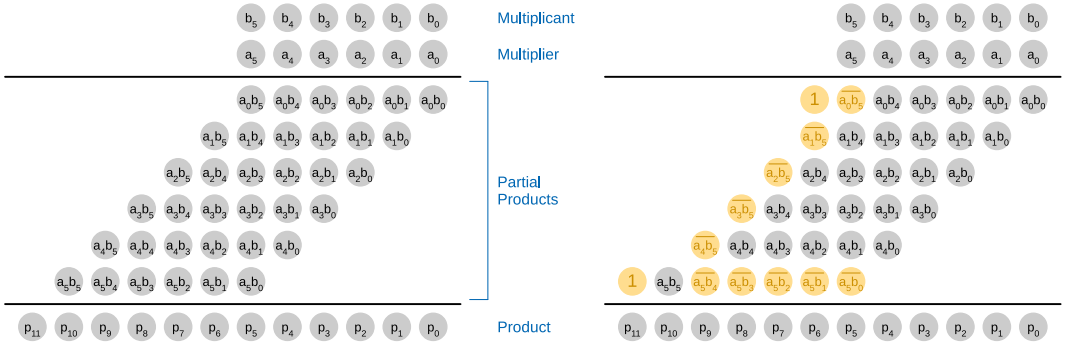
Multi-Segment Adders To achieve even more speed-up than possible with a segmentation into two parts, multi-segment carry-split adders were proposed. The ETA-II [55] splits an n-bit adder into m segments of size $\frac{n}{m}$. Each segment implements a carry generator that feeds the carry-in of the next higher segment and a sum generator that produces the sum bits related to the segment. Hence, the critical path delay behaves inversely proportional to the number of segments at the cost of errors generated at the carry split positions. Kahng and Kang proposed the *accuracy-configurable adder (ACA)* [56], which splits the addition into multiple sub-adder segments that each overlap by half their size, e.g. splitting a 16bit adder into three 8bit segments that overlap by 4 bits each. Their design additionally features an optional mechanism for error detection and correction which detects missed carries and increments the result accordingly while stalling the execution for one additional cycle. Allowing a more flexible configuration, the *gracefully degrading adder (GDA)* [57] uses a scheme similar to ETA-II with a carry prediction and an addition block per segment. However, they use multiplexers to dynamically combine adders across segments into larger sub-adders and/or to combine multiple carry prediction blocks into longer ones. The functionality of the multiplexers can either be fixed or dynamically controlled during operation. Finally, Shafique et al. proposed a generalized methodology of generating adders with overlapping segments in which both the number of final result bits produced per segment and the length of the carry prediction per segment can be configured completely freely [58]. Thus, they cover a design space that includes and supersedes all multi-segment adders discussed above.

Evolved Approximate Adders In Section 2.1.2.3, various methods for automatic approximate logic synthesis for general-purpose circuits were discussed. Mrazek et al. utilized the CGP-based ALS method [38–41] to evolve 430 approximate 8-bit adders, which are published as part of a library called *EvoApprox8b* [59]. An extended version of the library called *EvoApproxLib* was published recently, adding 12-bit, 16-bit and 32-bit adders [60]. While these units provide favorable quality-resource trade-offs, they are not parameterizable in a straightforward way, and only specific sizes are available.

2.1.2.5 Approximate Multipliers

Generally, to multiply two binary numbers, an array of partial products is generated which need to be summed up to get the final product, as shown in Figure 2.5a. The partial products are usually formed by feeding pairs of input bits into an AND gate. Different architectures to accumulate the partial products exist, e.g. implementing a carry-save adder array or using an adder tree structure, such as a *Wallace tree* or a *Dadda tree* [103]. For multiplication of signed inputs in two's complement form, the partial product array can be modified as shown in Figure 2.5b, which represents the *modified Baugh-Wooley multiplier* [108], or Booth encoding can be used [103].

Different methodologies for the approximation of multiplication circuits can be found in the literature. *Approximate multiplication cells* were proposed, which simplify the generation of the partial products. Secondly, *array truncation multipliers* perforate the partial product array to reduce resource usage. *Significance-based multipliers* dynamically extract significant portions of the input signals and feed them into a reduced-size core multiplier. Furthermore, *rounding-based multipliers*



(a) Unsigned multiplication

(b) Signed multiplication using modified Baugh-Wooley multiplier [108]

Figure 2.5: Partial product array of (a) unsigned and (b) signed Baugh-Wooley multiplication (adapted from [103])

and *logarithmic multipliers* use two different approaches to replace the multiplication by at most two adders and a few dynamic shift operations. Lastly, the libraries of evolved adders mentioned in the last section also contain *evolved multipliers*. Selected methods across these categories are summarized in Table 2.3 and discussed in the following paragraphs.

Approximate Multiplication Cells This group represents approaches that propose small multiplication cells, i.e. building blocks, from which larger multipliers can be built.

An early work by Kulkarni et al. proposes a 2×2 underdesigned multiplier (UDM) block, which simplifies the circuit by changing the output of $11 \cdot 11$ from 1001 to 111, effectively reducing the number of output bits to three for all input combinations [61]. This leads to a maximum error magnitude of 2 which occurs in $\frac{1}{16}$ of all cases.

Rehman et al. published *lpACLib*, a library of low-power approximate computing modules, which features a new simplified 2×2 multiplication building block that changes the truth table for three input combinations, but leads to a maximum deviation of 1 in all cases [62]. Hence, a different trade-off with a higher error rate but lower maximum error compared to the UDM block is achieved. Their methodology uses the new block together with the UDM and an accurate 2×2 multiplier together with several approximate adder designs to form a design space for larger multipliers which is searched greedily to find the optimum solution given a specified quality threshold.

Specifically targeting FPGA designs, Ullah et al. propose approximate 4×2 multiplier blocks which after truncating the LSB can be mapped efficiently to the available LUT architecture of Xilinx 7-series FPGAs [63]. Building upon these blocks, basic 4×4 blocks are formed, which in turn are used to build larger multipliers, for which two options are proposed. The first is the *Ca* architecture, which adds the partial products accurately and the second is called *Cc* which removes the carry propagation completely. While this leads to a considerable reduction of latency, it leads to significant increases in error magnitude.

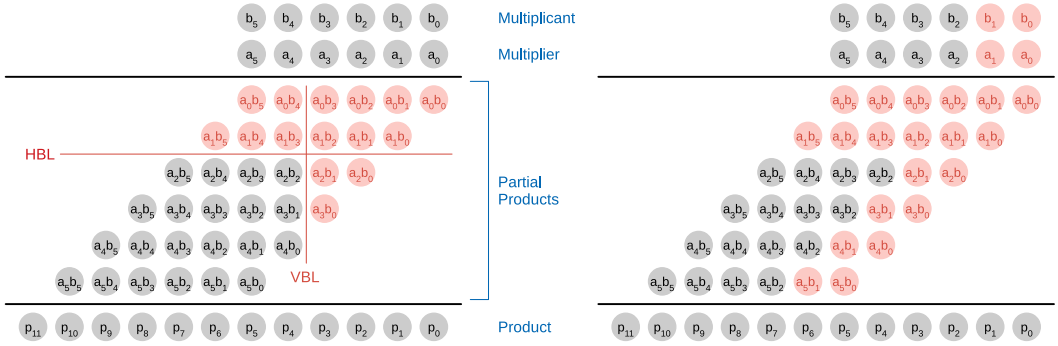
Approximate Array Multipliers The second group of approximate multipliers achieves a reduction in area and latency by truncating the partial product array. Mahdiani et al. proposed the idea of the broken-array multiplier (BAM) [50], which introduces a horizontal break level (HBL)

Table 2.3: Overview of approximate multipliers

Abbreviation	Full Name / Description	Reference
	Approximate Multiplication Cells	
UDM	Under-Designed Multiplier	[61]
lpACLib	Library of Low-Power Approximate Computing Modules	[62]
—	Area-Optimized Low-Latency Approximate Multipliers for FPGAs	[63]
	Array Truncation Multipliers	
BAM	Broken-Array Multiplier	[50]
BBM	Broken-Booth Multiplier	[64]
	Significance-Based Multipliers	
ETM	Error-Tolerant Multiplier	[65]
SSM	Static Segment Multiplier	[66]
DRUM	Dynamic Range Unbiased Multiplier	[67]
	Rounding-Based Multipliers	
ROBA	Rounding-Based Multiplier	[68]
RBA	Rounding-Based Multiplier (with more configurations)	[69]
	Logarithmic Multipliers	
LM	Basic Logarithmic Multiplier according to Mitchell's algorithm	[70]
Mitch-w	Truncated Logarithmic Multiplier	[71]
MBM	Minimally Biased Multiplier	[72]
REALM	Reduced-Error Approximate Logarithmic Multiplier	[73]
ALM	Approximate Logarithmic Multiplier	[74]
ILM	Improved Logarithmic Multiplier	[75]
	Evolved Multipliers	
EvoApprox8b	Library of evolved 8-bit adders and multipliers	[59]
—	16-bit multipliers formed by combining multipliers from EvoApprox8b	[76]
EvoApproxLib	Extended library of evolved adders and multipliers of various sizes	[60]

and a vertical break level (VBL) which control the truncation of rows and, respectively, columns of the partial product array. Figure 2.6 illustrates how this approach differs from directly truncating the bitwidth of the operands. Between both parameters, the HBL acts as a coarse configuration of the quality-resource trade-off, while the VBL is used for fine-grained adjustments. Inspired by the BAM, the broken-Booth multiplier (BBM) carries over the same principle and applies it to a Booth multiplier structure [64]. However, the BBM uses only the VBL as configuration parameter.

Significance-Based Multipliers The main principle of the significance-based multipliers is to dynamically select the significant portions of the input operands depending on the leading-one position and feeding them into a reduced-size core multiplier. An early approach, the *error-tolerant multiplier (ETM)* [65] splits the operands into an upper and a lower part. If there are only zeros in both upper parts, then the lower parts are fed into a small accurate multiplier. Otherwise, the upper



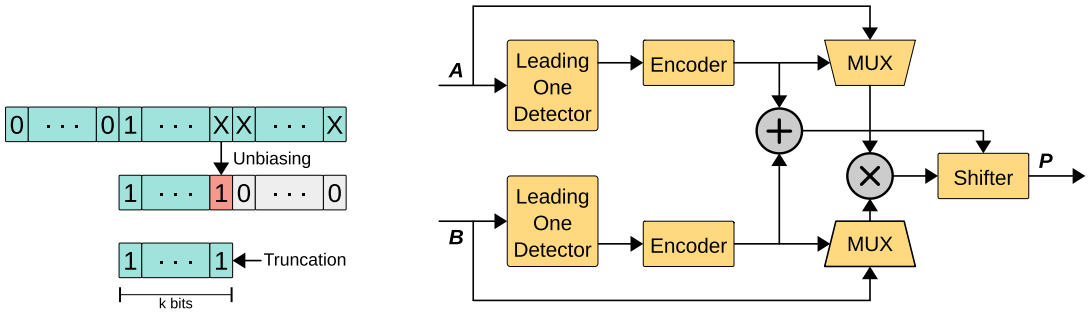
(a) Truncated partial product array of BAM (b) Multiplication with truncated input operands

Figure 2.6: Array truncation of the BAM compared to direct truncation of input operands (removed partial products are indicated in red shading, (a) adapted from [50])

parts are fed into the accurate multiplier and the multiplication of the lower parts is approximated using a scheme similar to the ETA-I, where from left to right each result bit is computed as the OR of the respective input bits and if both inputs are `1`, then all result bits to the right are set to `1`.

The *static segment multiplier (SSM)* [66] has a more flexible structure than the ETM. It splits each input operand into two (normal mode) or three (enhanced mode) segments of size m, which may overlap. In contrast to the ETM, the most significant portion is selected individually for each operand and forwarded to a smaller, accurate $m \times m$ multiplier. The result then has to be shifted based on the positions of both chosen segments.

To enable even finer adjustments, Hashemi et al. proposed the *dynamic range unbiased multiplier (DRUM)* [67], which removes the restriction to a small number of static segments and dynamically selects the most significant portion of both inputs directly from the respective leading-one position. Additionally, to unbias the approximation error, the LSBs of the extracted portions are set to `1`, representing the expected value of the truncated LSBs of the operand, assuming a uniform distribution in the inputs. This dynamic extraction and unbiasing from an input is depicted in Figure 2.7a and the resulting structure of the multiplier is shown in Figure 2.7b. Because of the highly flexible selection and the bias removal, DRUM is more accurate than ETM and SSM, but the selection and control circuitry is also the most complex among the three.



(a) Extraction and unbiasing (b) Multiplier structure

Figure 2.7: DRUM input processing (a) and implementation structure (b) (adapted from [67])

Rounding-Based Multipliers The *rounding-based multiplier (RoBA)* [68] replaces the multiplication by multiple shift and add operations after rounding the operands to the nearest powers of two. Given two input operands A and B and the respective rounded versions A_R and B_R, the multiplication can be expressed as

$$A \cdot B = A_R \cdot B + B_R \cdot A - A_R \cdot B_R + \underbrace{(A_R - A) \cdot (B_R - B)}_{\text{neglected term}}, \tag{2.4}$$

where the first three terms can be implemented by shift operations. The final term, which can be thought of as a rounding error, is hard to compute but contributes only a small amount to the final result. Hence, it is ignored in the computation of the approximate result. The resulting multiplier structure is depicted in Figure 2.8.

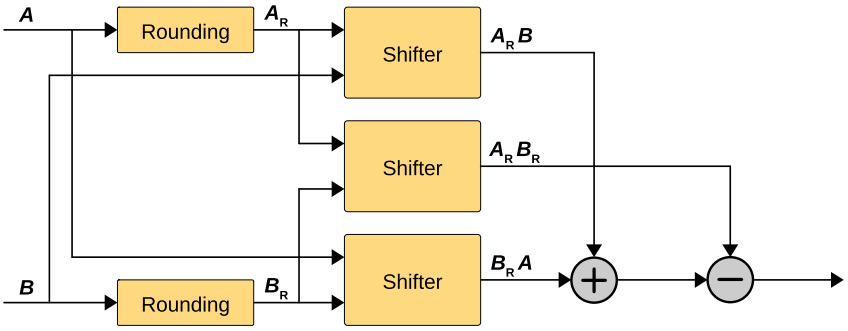


Figure 2.8: RoBA implementation structure for the unsigned case (adapted from [68])

Building upon the RoBA, Garg and Patel proposed to simplify the multiplication even further by using fewer terms of Equation 2.4 to form the variants RBA0 to RBA3 [69]. While RBA3 is the same as the original RoBA, the others are using the following equations:

$$\text{RBA0:} \quad A \cdot B = A_R \cdot B_R, \tag{2.5}$$

$$\text{RBA1:} \quad A \cdot B = A_R \cdot B, \tag{2.6}$$

$$\text{RBA2:} \quad A \cdot B = \left(A_R \cdot B + B_R \cdot A\right) \div 2, \tag{2.7}$$

offering a further reduction in implementation complexity at the cost of increased errors.

Logarithmic Multipliers This group of multipliers performs the operation as plain addition in logarithmic domain and uses approximate logarithmic conversion according to Mitchell's algorithm [70]. In this algorithm, the input operands are expressed as [109]

$$A = 2^{k_A}(1 + x_A), \tag{2.8}$$

$$B = 2^{k_B}(1 + x_B), \tag{2.9}$$

where k_A and k_B are the position of the leading-one in A and B, also referred to as *characteristics*, and $0 \leq x_A, x_B < 1$ are the fractional mantissa parts extracted from the right of the leading-one in A and B. The product of A and B is then given as

$$P = A \cdot B = 2^{k_A + k_B}\left(1 + x_A\right)\left(1 + x_B\right), \tag{2.10}$$

and its base-2 logarithm is

$$\log_2(P) = k_A + k_B + \log_2(1 + x_A) + \log_2(1 + x_B). \tag{2.11}$$

Using the approximation

$$\log_2(1 + x) \approx x \tag{2.12}$$

which holds for $0 \leq x < 1$, Equation 2.11 can be simplified to

$$\log_2(P) \approx k_A + k_B + x_A + x_B \tag{2.13}$$

and the final product in linear domain can be calculated by

$$P \approx \begin{cases} 2^{k_A+k_B}\left(x_A + x_B + 1\right), \text{if } x_A + x_B < 1, \\ 2^{k_A+k_B+1}\left(x_A + x_B\right), \text{if } x_A + x_B \geq 1. \end{cases} \tag{2.14}$$

The basic structure of a logarithmic multiplier (LM) is depicted in Figure 2.9. First, the leading-one is found and its position encoded as binary number $k_{A,B}$. Then, the logarithmic converter concatenates $k_{A,B}$ and the mantissa parts $x_{A,B}$ which are extracted from the right of the leading-one. The resulting pseudo-logarithmic numbers are added and the result is converted back to linear domain according to Equation 2.14. This step is done by the antilogarithmic converter by splitting the adder result into a characteristic and a mantissa part, setting the leading-one bit in the result word according to the characteristic part and appending the mantissa part to the right of it.

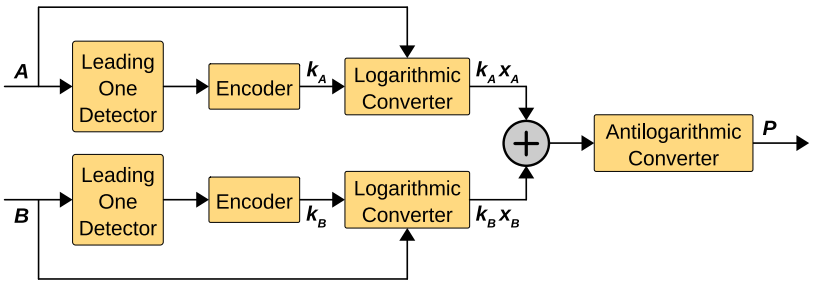


Figure 2.9: Implementation structure of basic LM

Kim et al. proposed to add a set-zero component to the LM to ensure a correct output if one of the operands is zero [71]. Furthermore, they propose to truncate the mantissa to a configurable width which lowers the resources needed for the logarithmic conversions as well as the adder at the cost of additional errors, allowing to flexibly control the resource-quality trade-off.

In the basic LM, the error scales with the power-of-two intervals of the input operands, determined by k_A and k_B. To reduce and unbias the error evenly across the input range, Saadat et al. proposed the *minimally biased multiplier (MBM)* [72]. Their architecture includes a constant error correction term that is added before the antilogarithmic conversion and consequently automatically scales correctly for the final output. The *reduced-error approximate logarithmic multiplier (REALM)* refines this idea by partitioning the power-of-two intervals into an $M \times M$ grid and storing different correction terms for each tile, addressed by the MSBs of the mantissa signals x_A and x_B [73]. At the cost of a correction coefficient table, this enables further reduction

of the output errors. Both the MBM and the REALM also add the possibility of truncating the length of the mantissa.

Liu et al. proposed several *approximate logarithmic multiplier (ALM)* variants, which employ different flavors of approximate addition [74] instead of an accurate adder. They show that employing a set-one adder (SOA), i.e. setting the output below a split point to constant `1`s leads to energy savings and slightly reduces the error as well.

Lastly, the *improved logarithmic multiplier (ILM)* [75] replaces the leading-one detector of the original LM design by a *nearest-one detector*. Hence, this method may choose the position left of the leading-one for the characteristic if the input operand is closer to the next higher power of two. This leads to a less biased logarithmic conversion at the cost of additional complexity.

Evolved Approximate Multipliers Similar to the evolved adders discussed above, also approximate multipliers were evolved using the CGP-based ALS method. In addition to the approximate adders, *EvoApprox8b* contains a large number of approximate unsigned 8×8 multipliers [59]. In [76], these multipliers are combined to form 16×16 multipliers. The recent *EvoApproxLib* library added several directly evolved unsigned $8 \times N$ multipliers together with signed and unsigned 8×8 and 12×12 multipliers [60]. Also, signed and unsigned 16×16 and 32×32 multipliers were added using the same divide-and-conquer strategy that was used in [76]. Similar to the evolved adders, these libraries contain units with favorable quality-resource trade-offs, but offer no straightforward parameterization, and feature only units of specific sizes.

2.1.3 Algorithm Level (Elementary Functions)

To distinguish different classes of approximations on the algorithmic level, we distinguish between approximating elementary functions, as presented in this section, and methods targeting the sequential control flow of an application (see Section 2.1.4). Elementary functions are small kernels that compute a given mathematical function, such as trigonometric, exponential/logarithmic or root functions, or compositions thereof. These functions have in common that it is virtually impossible to compute the exact results directly, and so they always need to be approximated in computational systems. However, normally implementations of such functions target the highest possible precision, i.e. trying for the result to be accurate up to the precision of the LSB of its numerical representation [110]. However, in the scope of approximate computing, this notion can be relaxed depending on the resilience of the target application and the effort spent of computing these functions may in turn be significantly reduced.

When implemented as part of a digital signal processing system, these kernels can be thought of as building blocks or sub-circuits that handle such computations in the datapath of the application. In the following, we give an overview of the most prominent methods of approximating elementary functions, focusing on how the quality-resource trade-off related with their implementation in hardware can be controlled meaningfully.

2.1.3.1 Polynomial Approximation

Polynomial approximation is a widely used way to implement complex math functions. As the name suggests, the accurate function is replaced by a polynomial expression of degree n that minimizes the average or worst case deviation between the polynomial and the accurate function over a target interval $[a, b]$. Methods of computing optimal polynomials are summarized in [111].

In hardware, polynomials can be implemented using multipliers and adders and the quality-resource trade-off can be controlled by choosing the polynomial degree n. However, the coefficients provided by standard algorithms are typically high-precision representations of real numbers which might not be suitable for hardware implementation and a plain quantization of these coefficients may lead to sub-optimal solutions [110]. To overcome this problem, Brisebarre et al. proposed *sparse-coefficient polynomials*, restricting the bitwidth of polynomial coefficients for efficient hardware implementation [77]. Chevillard et al. developed the *Sollya* tool, which can compute optimal polynomials with sparse coefficients given specific bitwidth constraints [78]. Going one step further, de Dinechin et al. proposed a heuristic method to find the best constraints that fulfill a given approximation error bound [79]. Their method is integrated in the *FloPoCo* tool which is used to automatically generate efficient arithmetic circuits for FPGA designs [112].

However, it should be noted that the success of polynomial approximation in achieving a favorable quality-resource trade-off, i.e. finding an acceptable solution with a low degree and small coefficients, varies strongly between different approximated functions and target intervals [110].

2.1.3.2 Incremental Refinement

Another method of approximating elementary functions is given in incremental refinement algorithms, i.e. algorithms which over repeated iterations improve the accuracy of the approximation. Here, we will focus on *shift-and-add* algorithms, which contain only additions and multiplications by powers of two, and are therefore well suitable for implementation in custom circuits. In the actual circuit design, the iterating kernel can be placed once and re-used by feeding the output back to the input until the target accuracy is satisfied. While this option allows for dynamic scaling of the output quality, it is not suitable for streaming processors that take in a new input in every cycle. Instead, for those cases, the iteration can be unrolled into a statically fixed number of subsequent copies of the kernel, which yields a fixed accuracy at the output. In the first case, the output quality is mainly traded off against computational performance, as more accuracy requires more time and hence throughput is reduced. Conversely, in the second case, quality is traded off mainly against area, as a higher accuracy requires more parallel instances of the iteration kernel (which likely also leads to a larger delay, but does not affect throughput in streaming processors).

A very prominent shift-and-add algorithm is the coordinate rotation digital computer (CORDIC) method, proposed by Volder in 1959 [80]. It performs a series of pseudo-rotations by predefined angles that get smaller with each iteration to approximate any target rotation angle. In the basic form, the iterations allow the calculation of trigonometric functions. Walther extended the algorithm to more elementary functions, including hyperbolic functions, logarithms, exponentials and the square root [81]. Generally, CORDIC-based computations converge linearly, i.e. $(n + 1)$ iterations are needed to achieve n accurate bits in the result, but many advanced modifications have been proposed over the years to achieve better trade-offs [82].

2.1.3.3 Table-Based Methods

Instead of implementing the computation of an elementary function directly in hardware, it can be pre-calculated at the target precision and stored in a table which is accessed during runtime. This effectively trades computation effort for the memory needed for tabulation. However, if values are stored for all possible inputs, the memory consumption scales exponentially with the number of input bits, leading to high amounts of memory consumption. Arguably the most straightforward

way to reduce memory consumption is to address the table only with the first k bits of the input, resulting in a sparse table with uniform segmentation [110]. However, this method may not be optimal in terms of the quality-resource trade-off, as the segmentation is globally constraint by the segment with the worst-case deviation.

Various methods have been proposed to alleviate this problem, which can roughly be categorized as multiplier-less *table-add methods* and *piecewise polynomial methods*, which are described below. In general, table-based methods have the advantage that they are not restricted to specific elementary functions but can easily be adapted to arbitrary functions. By choosing a suitable segmentation and/or interpolation scheme, the quality-resource trade-off can be controlled effectively at fine granularity.

Table-Add Methods To increases the accuracy of quantized tabulation, the bipartite table method employs a multiplier-less linear interpolation within the segments [83, 84]. To do this, the input word x is split into three parts x_0, x_1, x_2 (of lengths n_0, n_1, n_2, respectively), where $x = x_0 + x_1 + x_2$, which are used to access two tables in parallel: a table of initial values (TIV) that returns a base value $a(x_0, x_1)$ and a table of offsets (TO) which yields an offset value $b(x_0, x_2)$ These values are added together to approximate a target function $f(x)$ as

$$f(x) = f(x = x_0 + x_1 + x_2) \approx a(x_0, x_1) + a(x_0, x_2). \tag{2.15}$$

More specifically, the TIV is addressed with the $n_0 + n_1$ MSBs of the input and typically stores the values of f at the midpoints of the $2^{n_0+n_1}$ segments. On the other hand, the TO is addressed by the n_0 MSBs and the n_2 LSBs of the input, and returns different approximation curves for each of the 2^{n_0} major sections. Typically, the approximation curve is linear. In that case, the size of the second table can be cut in half and the missing values reconstructed by symmetry [84]. Figure 2.10 illustrates the curve approximation using the symmetric bipartite table-add method.

To achieve further refinements, multipartite table methods were proposed, which use multiple TOs, addressed by different portions of the input word [85, 86]. This generally leads to smaller tables at the cost of more additions.

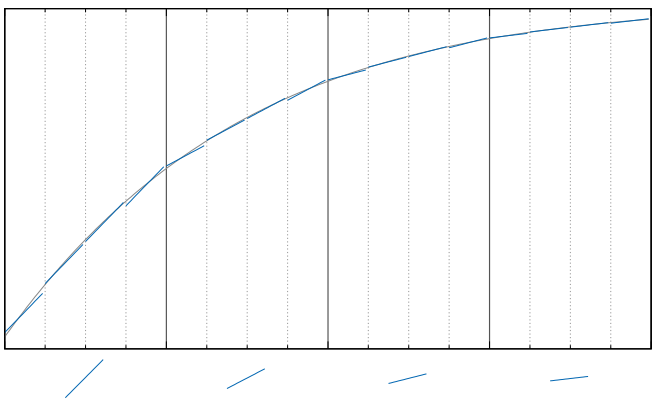


Figure 2.10: Curve approximation with symmetric bipartite table-add method

Piecewise Polynomial Methods The other type of table-based methods uses one or more multipliers to perform polynomial interpolations within the gaps of sparse tables [111]. In general, the MSBs of the input are used to address a table yielding interpolation coefficients while the LSBs are forwarded as input to the polynomial. For a polynomial degree of d, $d+1$ coefficients need to be stored. Polynomials of higher degrees can be decomposed according to the Horner rule to avoid costly power units, which requires d multipliers and d adders for polynomials of degree d. As a special case, for polynomials of degree 1, i.e. linear interpolation, the slope can either be stored in the table or derived from two consecutive table entries, but the latter requires dual-port access to the memory storing the table.

As mentioned above, uniform segmentation of the input space may lead to sub-optimal resource usage. To solve this problem, Lee et al. proposed a non-uniform, hierarchical segmentation scheme [87]. Their approach employs multiple levels of segmentation. At the highest level, the input interval is divided into several sections. Then, each of these sections is divided further into a variable number of lower-level segments, and so on. At each level, the segmentation can follow one of four schemes: either dividing into uniform segments or dividing into segments of exponentially changing sizes. When restricting the segment sizes to powers of two, the resulting addressing scheme can be implemented with little overhead in hardware. Figure 2.11 provides an example for a curve reconstruction using two uniformly distributed segmentation levels and linear interpolation.

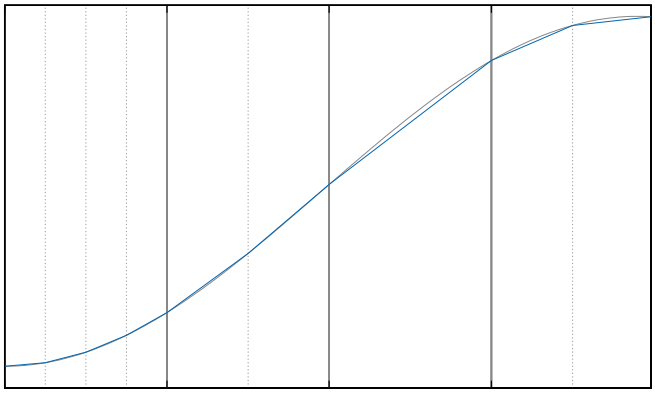


Figure 2.11: Curve approximation with two-level (uniform/uniform) hierarchical segmentation using 4 major sections with [4, 2, 1, 2] segments and linear interpolation within the segments

2.1.4 Algorithm Level (Control Flow)

At a higher abstraction within the algorithm level, there are several techniques which target the control flow, aiming at a reduction of necessary computations. In software, *loop perforation* [113] is a popular, dynamically scalable approach which intelligently skips selected executions in loops or other repeating tasks, e.g. in image processing algorithms [114]. Applicable to both software and hardware-based systems are the approaches of *fuzzy memoization* and *neural acceleration*, which are described below.

2.1.4.1 Fuzzy Memoization

Memoization describes a technique, where the results of executed instructions are stored in a cache-like table which is addressed by the instruction type and the operands [115]. When the same combination of instruction and operands is encountered again, the result can be read directly from the table and the computation can be avoided. While this technique is able to reduce power consumption, the size of the memoization table quickly grows with larger operands, making it less effective.

Therefore, Alvarez et al. introduced the idea of *fuzzy memoization*, which relaxes the condition of requiring identical inputs to access the table [88]. More specifically, by removing n LSBs from the operands before addressing the table, a range of similar inputs will also successfully hit the memoization table. Hence, this leads to a configurable trade-off between the power reduction, which improves with increasing hit rates, and output quality, which falls if larger tolerances are allowed.

Directly targeting implementation on FPGA-based systems, Sinha and Zhang propose a generalized methodology to create both static and dynamic memoization architectures automatically [89]. They allow the user to provide an arbitrary similarity measure and a tolerance threshold, defining the range of accepted similar inputs. In the static case, the memoization table is pre-filled with results for the most likely inputs in contrast to the dynamic case, which allows updating the memoization table during runtime. Their experiments show that at the cost of the area overhead introduced by the memoization control and memory, the power consumption can be improved compared to other approximation techniques, hence offering different quality-resource trade-offs depending on the design targets.

2.1.4.2 Neural Acceleration

In 2012, Esmaeilzadeh et al. proposed to accelerate general-purpose programs using neural processing units (NPUs), i.e. energy-efficient dedicated hardware to accelerate neural networks [90]. The main idea behind this is to convert entire blocks of imperative code into a neural network structure that can be called from the main program. Similarly, the idea could be applied for hardware-based systems to offload complex computations that would consume a high amount of logic resources into a specialized neural accelerator. If a good network topology is chosen and the network is trained appropriately, it can approximately mimic the accurate operation with manageable errors at the output. For demonstration purposes, they simulated a heterogeneous digital CPU-NPU architecture to estimate the potential benefits for several benchmark applications, and report an average speed-up of 3.7× and an average energy reduction of 3.0× at an average error of 6.87% across all applications. However, their results remain theoretical and vary largely between the applications. This concept was extended by Amant et al. to include an analog NPU in place of the digital one [91]. Their simulations show that the analog design improves the speed-up and energy savings significantly, but at an additional loss of accuracy. In 2015, *SSNAP* was proposed, which presents a real implementation of the concept in an off-the-shelf system-on-chip (SoC) FPGA platform [92]. While the main program runs on the CPU of the SoC, the NPU is implemented in the FPGA fabric. Reporting an average speed-up of 3.8× and an average energy reduction of 2.8×, their results show that their real-world measurements roughly matches the theoretical benefits.

2.1.5 Memory Level

Besides the core processing components, memory units are important components in computational systems. In FPGA-based devices, a small amount of near-processing memory is typically available in the form of random access memory (RAM) which is embedded alongside the logic fabric of the FPGA chip, and commonly abbreviated as BRAM. However, since the storage provided by the BRAMs is very limited, external memory modules are used to hold larger amounts of data. In FPGA-based digital camera systems, for example, the internal BRAM may hold a couple of lines of an image locally in row buffers while external RAM can be used to store multiple entire images in frame buffers [116]. Since memory units contribute a significant amount of power consumption [97], a range of techniques was proposed to approximate data storage to obtain benefits, mostly in power consumption. They are summarized in the sections below.

2.1.5.1 Memory Voltage Overscaling

Similar to the approaches discussed above in Section 2.1.1, where the supply voltage of computational circuits is overscaled to reduce power consumption, several methods have been proposed to apply aggressive overscaling in memory supply voltage in a controlled manner.

Chang et al. proposed to use a hybrid static RAM (SRAM) array consisting of a mixture between 6T and 8T cells to alleviate the effect of VOS applied to the memory [93]. Exploiting the fact that 8T cells behave more robustly at reduced supply voltages than conventional 6T cells, they store higher order bits in 8T cells and the lower order bits in 6T cells. The quality-power trade-off can then be controlled by adapting the 6T/8T cell ratio, which determines the amount of data that can be stored accurately, as well as by scaling the supply voltage, which influences the error probability of inaccurately stored bits. Instead of mixing different memory cells, *Truffle* proposes a novel dual-voltage architecture, in which the storage/retrieval logic of an SRAM array can dynamically be switched between nominal and reduced voltage to enable accurate and approximate operation of the same memory cells [94]. To control the operation mode, instruction set architecture (ISA) extensions are proposed to enable switching at the software/compiler level.

Targeting emerging spintronic memories, Ranjan et al. designed a quality-configurable memory array to enable data storage at varying quality-power trade-off levels [95]. To achieve this, they use three scalable mechanisms to approximate the operation of spin transfer torque magnetic RAM (STT-MRAM) cells: lowering the voltage and sense current in read operations, reducing the read duration at an increased voltage, or lowering the voltage and/or duration during write operations.

In 2018, Salami et al. studied the effects of scaling the supply voltage on the V_{CCBRAM} rail that globally powers the embedded BRAM memory units in FPGAs [96]. They found that BRAM can be reduced significantly below the nominal value without faults occurring, and even more when some faults are acceptable. However, their experiments show that there are significant die-to-die variations between sampled chips when operated below the nominal voltage. Additionally, they found non-uniform within-die variations in the behavior of different BRAMs on the same chip, which appear to be stable over time and deterministic. The error behavior can therefore be characterized per chip and stored in a fault variation map. This map can then be used to construct additional constraints for the placement stage to map less important data to BRAMs with higher fault probability to control the quality impact caused by overscaling the V_{CCBRAM} rail.

2.1.5.2 Refresh Rate Scaling

Dynamic RAM (DRAM) is a popular memory technology due to its low cost and high density. However, since DRAM is a non-persistent data storage technology, it needs to be refreshed periodically to prevent data loss, which consumes a considerable amount of power [117]. Decreasing the refresh rate alleviates the power consumption but may lead to erroneous data.

An early approach for controlling the resulting quality loss is called *Flikkr* [97], in which the application data is partitioned into a critical and a non-critical portion, which are stored in different sections of the memory. Then, the refresh control is modified to reduce self-refresh in rows that store non-critical data, resulting in a reduction of refresh power. Another technique called *Sparkk* [98] partitions the data within words by bit significance and maps the individual bits to different parts of the memory that operate at different refresh rates. This enables multiple quality levels to be employed for different bits which can lead to improved trade-offs at the cost of additional overhead introduced by the mapping and a more complex refresh control. Raha et al. proposed a methodology that exploits the fact that different pages of a memory module vary in their ability to tolerate a reduced refresh rate [99]. Extensive retention time measurements are used to sort the pages into different quality bins. This allows storing data at multiple quality levels while refreshing the entire chip at a single (reduced) rate, eliminating the need for modified refresh controls. However, this requires a time-consuming characterization phase for each DRAM module to be used and their results indicate that error characteristics may deviate largely between different modules.

2.1.5.3 Approximate Memory Operation

Besides voltage or refresh rate scaling, other methods have been proposed for approximate operation of memories to reduce power or even reactivate worn-out memory cells. Targeting solid state memories, Sampson et al. propose methods to approximate data storage in multi-level cell (MLC) memories [100]. MLCs store multiple bits in a single cell but require more time and energy to access and need additional error correction overhead. The cell data is stored as an analog value and guard bands are used to distinguish the different levels. To reduce the time and power consumed by access operations, the guard bands are relaxed which increases the probability for incorrect operation. Additionally, to extend the lifetime of solid state devices, they enable the continued use of worn-out blocks with, i.e. blocks with too many defective bits, by prioritizing the available error-correction resources on the most significant bits and allowing errors in the remaining ones.

Ganapathy et al. propose to alleviate error correction overhead in SRAM cells [101]. They replace the costly row-wise error-correction circuitry with a mechanism that shuffles the data on read/write accesses so that always the least important bits end up in a faulty cell, as guided by a fault-map table. This leads to reductions in area, power and delay at controlled quality degradation.

2.1.5.4 Memory Access Scaling

One method at the intersection of precision scaling and approximate storage is called *approximate memory access (approxMA)* [102]. Instead of reducing the bitwidth of on-chip signals and related operations, it scales the precision of words read from memory to reduce the necessary number of read operations. To enable this, the data is reorganized so that bits of similar significance from multiple words are stored together. A runtime precision controller then dynamically determines the required precision for a read operation and only loads the needed number of significant bits.

2.2 Selection and Implementation for FPGA

From the large range of approximation techniques reviewed in the previous section, we implemented several methods that are promising candidates for use in low-level image processing applications on FPGA hardware, focusing on reducing the resource consumption as a primary design target. The selection was based on the predominant operations found in such systems. First, we apply *precision scaling* which is a universal technique applicable in the design of any signal processing system. Furthermore, most image processing applications feature many *arithmetic operations*, e.g. for color calculations or filtering operations. Hence, we selected different approximate adders and multipliers to include within our library of approximate components. Lastly, image processing pipelines in digital cameras often contain point operations [118] which apply transfer functions to pixel values. Point operations may for example be used for luminance encoding or tone-mapping tasks and commonly feature non-linear functions. In order to support approximation for arbitrary transfer functions, we selected a flexible *table-based method* as well.

With the exception of precision scaling, the approximation methods can be thought of as approximate components or building blocks which can be combined with accurate components to form various applications. On the other hand, precision scaling affects the width of the connections between these components. Nevertheless, all selected methods are suitable for use in FPGA devices across different vendors, exhibit parameters which can be used to scale the degree of approximation, and influence the processed data deterministically. While for some methods, the implementation for FPGA designs is straightforward, others might need some adaption when transferring from descriptions targeting ASIC hardware.

The following sections describe the selected techniques in detail and motivate their choice. Furthermore, we provide implementation details specific to the meaningful use within typical FPGA architectures.

2.2.1 Precision Scaling

We use fine-grain precision scaling to adapt the width of individual signals within the data flow of the application. For this, the designer may freely select multiple signals and individually define the range in which their bitwidth can be scaled. In the hardware implementation, the scaling can easily be controlled via generics at the hardware description language (HDL) level. From a system perspective, it could also be considered as a component placed on a signal that removes a parameterizable number of bits from the signal. The most apparent impact of this is that the number of registers needed in the respective signal paths in pipelined systems directly scale accordingly. Another effect is that it changes the width of inputs and outputs for operations and components within the application, hence indirectly influencing their resource consumption as well. Nevertheless, this leads to a number of considerations to be made when precision scaling is combined with other approximation methods. For one, the other components need to be able to flexibly scale in size in order to carry over the benefits achieved with precision scaling. On the other hand, there may be interplay between the parameterizations of components and the bitwidth of signals carried between these components that needs to be taken into account to ensure a valid, meaningful global parameterization. While the first point influences the choice of components below, the second will be discussed in detail in Section 3.2 in the next chapter.

2.2.2 Approximate Adders

The logic architectures of current commercial FPGA platforms directly integrate dedicated fast carry chains to support fast addition, enabling the implementation of adders with low delay directly in the logic fabric. Using these dedicated resources for the accurate parts, we implemented all two-segment carry split adders shown in Figure 2.4, namely the LSA, the MA, the LOA, the sloppy adder, the OLOCA and the HOAANED. All of these adders are easily configurable by setting their overall input size as well as the split position, counting from the LSB. The LSA additionally uses an additional parameter to select which input is forwarded in the lower part.

When synthesized for FPGA designs, the input signals are fed through LUTs into the dedicated fast carry chains in the accurate part. However, the resource utilization of the approximate part depends on the adder type. LUTs are used throughout the approximate part of the LOA and the sloppy adder as well as the two most significant approximate bits of the OLOCA and the HOAANED, because these bits are generated by logic functions. In case of the LSA, the lower part only needs wires to connect the selected input through to the output. For the MA, the OLOCA and the HOAANED, some or all of the output bits in the lower part are replaced by constants, for which neither LUTs nor wires are utilized. Note that using one of these three adders has further impact on the preceding components, since some of the input bits do not need to be calculated. Similarly, with the LSA, some of the lower bits of the input that was not selected do not need to be calculated.

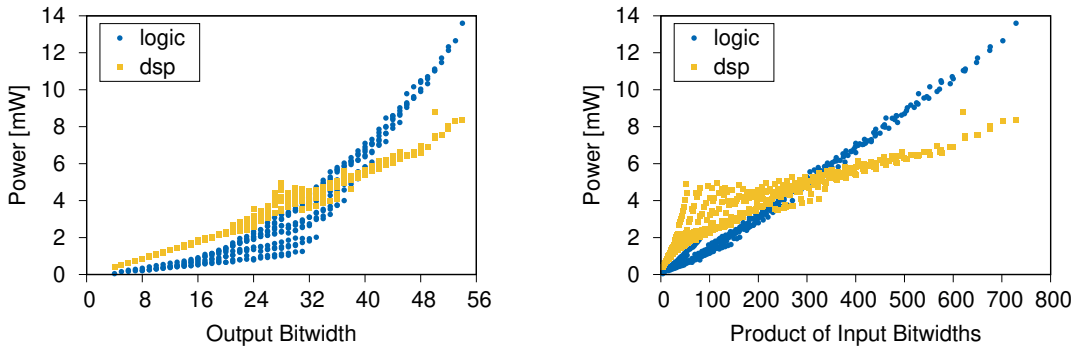
2.2.3 Approximate Multipliers

Similar to the fast carry chain structure embedded within the general logic fabric to accelerate addition operations, current FPGA platforms integrate dedicated hard DSP slices to facilitate efficient and fast multiplication. Typically, such DSPs can be used in various implementation modes, supporting different operator sizes. For example, the DSPs integrated in the Intel Arria 10 FPGA series in fixed-point mode support two independent 18×19 multiplications or one 27×27 multiplication.

Since the functionality of these hard multiplication circuits of the DSP slices cannot be altered, approximations can only be realized in the form of soft multipliers implemented in the logic fabric. Therefore, using approximate multipliers in FPGA designs is only beneficial in specific cases. First, logic multipliers may need to be used in large designs as the number of DSP slices available is limited and may be exhausted. Secondly, even though the dedicated multiplication circuitry is highly optimized, small multipliers can be more efficient when implemented in logic. To highlight the second point, Figure 2.12 compares the power consumption of logic multipliers and DSP multipliers across all possible size combinations between 2×2 and 27×27, directly inferred from behavioral description, on an Intel Arria 10 FPGA device. The data points were collected using the automatic component characterization methodology described later in Section 2.3. Shown are projections of power consumption against (a) output bitwidth, i.e. the sum of both input bitwidths, and (b) the product of both input bitwidths.

The plots show that the power consumption of DSPs scales roughly linearly with the output bitwidth but for logic implementations it rather scales linearly with the product of input bitwidths. Furthermore, for smaller sizes, multiplication in core logic is more power efficient than using DSP slices. We conclude that for multiplier sizes up to 18×18, logic multipliers and hence also approximate multipliers are of particular interest for FPGA designs.

Several state-of-the-art approximate multipliers have been selected from the literature and were ported into a corresponding FPGA design. In the selection, several factors were considered. First,



(a) Projection of power over output bitwidth

(b) Projection of power over product of input bitwidths

Figure 2.12: Comparison between logic implementation and DSPs in terms of power consumption for multipliers of sizes between 2×2 and 27×27

the implementation should be parameterizable to cover multipliers of arbitrary sizes. This excludes already many of the architectures proposed in the related literature which are restricted in some form or another to very specific size combinations. In most cases, the architectures require both inputs to be of the same bitwidth and many of them further restrict that bitwidth to be exactly a power of two. While it is possible to use a larger multiplier to accommodate a mid-size multiplication, it is not considered because it likely wastes resources unnecessarily. Secondly, we only consider architectures that offer further parameterization to control the degree of approximation. Lastly, even though we consider implementation on Intel Arria 10 series FPGA devices throughout this work, we want to consider only vendor-agnostic units, i.e. units that may synthesize regardless of the target platform and hence do not use vendor-specific primitives in their design. Unfortunately, this excludes virtually all proposed designs directly targeting FPGA designs, as they all exploit very specific aspects of a single target architecture and use vendor-specific primitives to access them [63].

With these considerations in mind, we chose to port the BAM [50], the DRUM [67], the RoBA [68] with an optional truncation for internal signals together with the additional operation modes that were proposed as RBA [69] and the logarithmic Mitch-w [71]. Several adaptions had to be made for the implementation on FPGA hardware, which are explained in the following sections.

2.2.3.1 Broken-Array Multiplier (BAM)

For the BAM, a custom implementation of the multiplication array and the subsequent partial product summation is needed so that generic truncation at the horizontal (HBL) and vertical (VBL) break levels can be realized. First, the partial product array is formed as in Figure 2.5a by taking the AND of all bit pairs from the inputs A and B. For signed multiplication, the modified Baugh-Wooley structure [108] is used as illustrated in Figure 2.5b. The summation of partial products is implemented using a hybrid adder tree structure. For most of the adder tree, ternary adders are used to reduce the resource consumption as they can be efficiently implemented on both Xilinx and Intel FPGAs using the LUTs and dedicated carry chains available in the logic architecture. The first layer, however, uses the LUTs within the logic blocks to create the partial products and hence performs binary addition to add two consecutive rows of the array within the dedicated carry

chain. If the input operands are of unequal size, the smaller operand is taken as multiplier, which determines the number of partial product rows, to minimize the depth of the adder tree.

To truncate the partial product array in a generic manner according to the HBL and VBL parameters, the respective partial products are forced to constant *0*s and the addition operations are shortened accordingly. This implementation supports arbitrary input size combinations. The HBL scales up to the width of the smaller operand whereas the VBL can take values up to the output width. In the case of signed multiplication (cf. Figure 2.5b), the truncation needs to omit the modified array cells to ensure meaningful operation.

2.2.3.2 Dynamic Range Unbiased Multiplier (DRUM)

The basic operation principle of the DRUM is to detect the leading-one in both inputs and extract a defined number of bits after it to be fed into a smaller core multiplier. This input processing involves three steps: Leading-one detection, numerical encoding of its position, and extracting the k subsequent bits from the input, where k is the core size. In the original publication, this is done in sequence, resulting in a significant delay overhead. Our implementation uses a self-determined shifter, which shifts the input to the left in power-of-two steps until the leftmost bit becomes `1`. Each shift layer simultaneously records the executed power-of-two shifts in an output signal, which directly coincides with the encoded leading-one position. Hence, all three steps can be implemented within one functional block, as shown in Figure 2.13. When unequally wide input operands are used, the range for the core size k scales up to the width of the smaller operand.

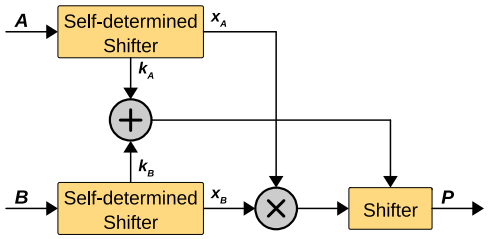


Figure 2.13: Adapted unsigned DRUM structure using self-determined shifters which simultaneously produce the leading-one positions $k_{\mathrm{A,B}}$ and the extracted parts $x_{\mathrm{A,B}}$ from the inputs A, B

For the handling of signed computation, the original paper suggested enclosing the proposed method, which only works for the unsigned case, between additional pre- and post-processing logic that converts from two's complement to unsigned format and vice versa. Generally, this conversion is done accurately by inverting all bits and adding 1 to the result in both directions, which implies the use of two extra adders. To circumvent the associated additional delay, several works propose to use an approximate conversion by removing the addition, introducing an additional error of one code value in each conversion [68, 71]. We implemented a conversion wrapper that supports both accurate and approximate operation which can be reused for all multipliers whose core architecture supports unsigned operation only, as shown in Figure 2.14.

2.2.3.3 Rounding-Based Approximate Multiplier (RoBA) and variants

Recount that the RoBA approximates the computation using the following formula:

$$A \cdot B \;=\; A_{\mathrm{R}} \cdot B \;+\; B_{\mathrm{R}} \cdot A \;-\; A_{\mathrm{R}} \cdot B_{\mathrm{R}}, \tag{2.16}$$

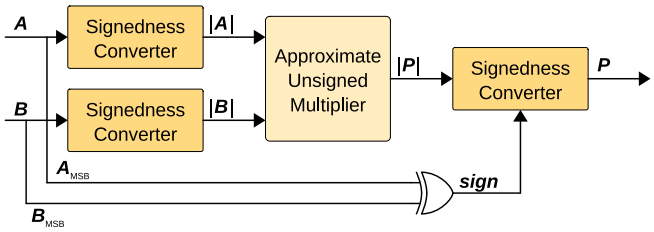


Figure 2.14: Signedness conversion wrapper to enclose unsigned approximate multipliers

where A, B are the inputs and A_R, B_R are the inputs rounded towards the next power of two. Hence, three shift operations as well as one addition and one subtraction are needed in the original design.

The operation can further be simplified by reformulating Equation 2.16 depending on the rounding direction. Let A respectively B be described as $2^{k_{\mathrm{A,B}}}(1+x_{\mathrm{A,B}})$, where $k_{\mathrm{A,B}}$ are the leading-one positions of the inputs A and B and $0 \leq x_{\mathrm{A,B}} < 1$ are the bits after the leading-one, interpreted as mantissa part. Then, the rounded numbers are:

$$A_\mathrm{R} = \begin{cases} 2^{k_\mathrm{A}+1} & \text{if } x_\mathrm{A} \geq 0.5, \\ 2^{k_\mathrm{A}} & \text{else,} \end{cases} \qquad B_\mathrm{R} = \begin{cases} 2^{k_\mathrm{B}+1} & \text{if } x_\mathrm{B} \geq 0.5, \\ 2^{k_\mathrm{B}} & \text{else.} \end{cases} \tag{2.17}$$

The value of the bit after the leading one decides between rounding up and rounding down.

Three cases can be distinguished, namely (a) both operands are rounded up, (b) both operands are rounded down, and (c) either one operand is rounded up while the other one is rounded down. By inserting the rounded values according to Equation 2.17 into Equation 2.16 for each of these three cases, we obtain the following simplifications:

(a) Both operands are rounded up:

$$\begin{aligned} A \cdot B &= A_\mathrm{R} \cdot B + B_\mathrm{R} \cdot A - A_\mathrm{R} \cdot B_\mathrm{R} \\ &= 2^{k_\mathrm{A}+1} \cdot 2^{k_\mathrm{B}} (1+x_\mathrm{B}) + 2^{k_\mathrm{B}+1} \cdot 2^{k_\mathrm{A}} (1+x_\mathrm{A}) - 2^{k_\mathrm{A}+k_\mathrm{B}+2} \\ &= 2^{k_\mathrm{A}+k_\mathrm{B}} (2x_\mathrm{A} + 2x_\mathrm{B}). \end{aligned} \tag{2.18}$$

(b) Both operands are rounded down:

$$\begin{aligned} A \cdot B &= A_\mathrm{R} \cdot B + B_\mathrm{R} \cdot A - A_\mathrm{R} \cdot B_\mathrm{R} \\ &= 2^{k_\mathrm{A}} \cdot 2^{k_\mathrm{B}} (1+x_\mathrm{B}) + 2^{k_\mathrm{B}} \cdot 2^{k_\mathrm{A}} (1+x_\mathrm{A}) - 2^{k_\mathrm{A}+k_\mathrm{B}} \\ &= 2^{k_\mathrm{A}+k_\mathrm{B}} (1 + x_\mathrm{A} + x_\mathrm{B}). \end{aligned} \tag{2.19}$$

(c) One operand (e.g. A) is rounded up, the other (e.g. B) is rounded down:

$$\begin{aligned} A \cdot B &= A_{\mathrm{R}} \cdot B + B_{\mathrm{R}} \cdot A - A_{\mathrm{R}} \cdot B_{\mathrm{R}} \\ &= 2^{k_{\mathrm{A}}+1} \cdot 2^{k_{\mathrm{B}}} (1 + x_{\mathrm{B}}) + 2^{k_{\mathrm{B}}} \cdot 2^{k_{\mathrm{A}}} (1 + x_{\mathrm{A}}) - 2^{k_{\mathrm{A}}+k_{\mathrm{B}}+1} \\ &= 2^{k_{\mathrm{A}}+k_{\mathrm{B}}} (1 + x_{\mathrm{A}} + 2x_{\mathrm{B}}). \end{aligned} \tag{2.20}$$

Similarly, if B is rounded up and A is rounded down, the result is given as

$$A \cdot B = 2^{k_{\mathrm{A}}+k_{\mathrm{B}}} (1 + 2x_{\mathrm{A}} + x_{\mathrm{B}}). \tag{2.21}$$

Therefore, if x_{A} and x_{B} are extracted from the inputs using self-determined shifters, the rest of the operation can be executed using two adders and another normal shifter. The first one determines the shift amount while the second one calculates the value to be shifted. In case (a), x_a and x_b are shifted to the left by one before their addition. However, in case (c), only either x_a or x_b is shifted. Additionally, in cases (b) and (c), a `1` is pre-pended to the left of one operand before the addition. In case (c), this applies to the operand that is not left-shifted while in case (b) either operand can be used for this. The adapted structure of the multiplier is shown in Figure 2.15.

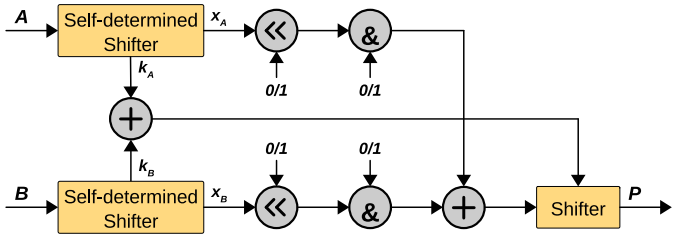


Figure 2.15: Adapted unsigned RoBA implementation structure where the mantissa parts $x_{\mathrm{A,B}}$ may be left-shifted by 1 bit (indicated by «) or left-appended by `0` or `1` (indicated by **&**), depending on the rounding mode (cf. Equations 2.18 - 2.21)

Furthermore, this implementation allows x_a and x_b to be variably truncated to a shorter width w to reduce the size of the shifters and the second adder, enabling parameterizable control over the achieved quality-resource trade-off. In addition to this parameterizable version of the main RoBA design, further denoted as RoBA-w, we implemented the simplified RBA0-RBA2 designs defined in Equations 2.5 - 2.7 (without truncation) for comparison.

2.2.3.4 Truncated Logarithmic Multiplier (Mitch-w)

For the truncated logarithmic multiplier, we follow the structure proposed for the original Mitch-w multiplier [71]. However, as for the variants of DRUM and RoBA described above, we employ the self-determined shifter block to extract the leading-one position and the truncated mantissa part simultaneously, which are then concatenated to form the approximate logarithms of the input, as shown in Figure 2.16.

2.2.4 Table-Based Methods

In order to approximate the evaluation of non-linear functions, we chose the hierarchical table segmentation method proposed by Lee et al. [87] because of its flexibility in adapting to arbitrary

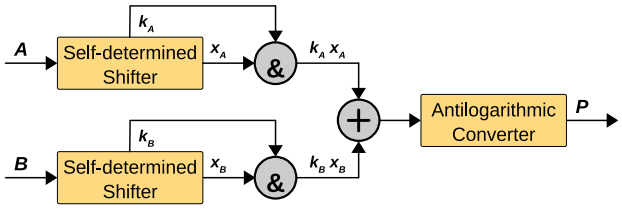


Figure 2.16: Adapted unsigned Mitch-w implementation structure using self-determined shifters (**&** indicates concatenation of signals)

functions. The segmentation scheme defines the distribution of grid points across the input range for which pre-calculated values are stored in a sparse table. Our implementation uses two hierarchy levels that both utilize a uniform distribution, as illustrated exemplarily in Figure 2.11. At the top level, the input range is divided into N_{sec} major sections. Then, each of these sections is split further into $N_{\text{seg}}(i)$ minor sub-segments, where i indicates the index of the respective major section. As another parameter, we optionally employ linear interpolation to improve the reconstruction of values in the gaps between the grid points.

For the practical implementation, we consider the input as an n-bit word and hence the input range covers 2^n values. If the values for N_{sec} and $N_{\text{seg}}(i)$ are also each restricted to powers of two, the segmentation can be done with minimal overhead. The internal structure of the *sparse table* component is shown in Figure 2.17.

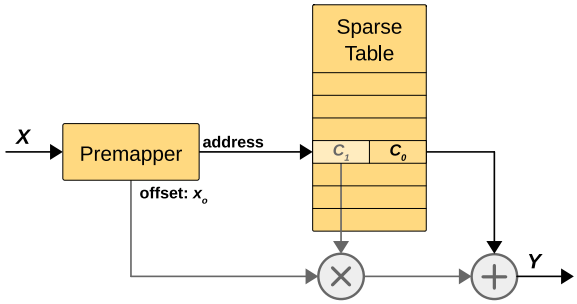


Figure 2.17: Implementation structure of hierarchically segmented sparse table (parts only needed when interpolation is used are shaded in gray)

A combinatorial *premapper* block calculates both the address to the sparse table and, if necessary, the offset beyond the current gridpoint, which is needed for the interpolation. Because the segmentation is uniform/uniform across both levels, the first $\log_2(N_{\text{sec}})$ bits directly determine the section i. Then, the next $\log_2(N_{\text{seg}}(i))$ bits are used to determine the segment within the section. The address is finally formed by adding the segment number to the cumulative number of segments contained in lower sections which is stored in a very small table directly in the logic fabric. On the other hand, the offset x_o is given by the remaining input bits that were not used for address calculation, shifted to a normalized width. Each word in the sparse table stores a base value C_0 and an interpolation coefficient C_1 used for the output reconstruction in the respective segment accessed by the calculated address. The reconstruction of the output is then calculated as

$$Y = C_1 \cdot x_o + C_0. \tag{2.22}$$

If the interpolation is turned off, the output is given directly by the coefficient C_0 without further calculation, i.e. the interpolation circuitry can be removed and the table does not need to store C_1.

2.3 Characterization

To better understand the specific resource-quality trade-offs afforded by different approximate components, they need to be characterized with respect to the chosen target FPGA architecture. The prominent implementation characteristics in pipelined hardware systems are area and power consumption, which are typically closely related, together with the circuit delay. For FPGA designs, the area consumption is measured in terms of the different available resource types such as LUTs, registers, BRAMs and DSP slices. The power consumption is composed of a static part that scales with the number of resource units and a dynamic part which additionally depends on the signal switching activity. Besides, the circuit delay is determined by the longest combinatorial path between two register stages and defines the maximum possible frequency at which the circuit can be operated. On the other hand, as a fourth property of the approximate component, the loss of accuracy caused by the approximation can be expressed in the form of error statistics calculated between the accurate and approximate output of the respective component when fed with specific input vectors.

A characterization of individual approximated components provides information on these metrics on a local level, i.e. independent of the particular application in which the components may later be used. It serves to compare competing techniques against each other, and, when targeting FPGA-based applications, shows how well different techniques adapt to the implementation on a specific FPGA architecture. Furthermore, the resulting data can be used to model the resource consumption across a multitude of possible parameterizations, as described later in Section 2.4.

The next section provides an overview of the automated characterization methodology used in this work to profile the quality-resource trade-off of approximate components. Using this methodology, the subsequent section provides an in-depth comparison and analysis of the approximate adders and multipliers selected and described in the previous section.

2.3.1 Methodology

As mentioned above, an approximate component implemented on FPGA hardware is characterized by four properties of interest, which are area, power, speed and accuracy loss. The first three of these can be classified as physical properties and depend on the specific FPGA device chosen for implementation together with environmental conditions. In contrast, the loss of accuracy is a behavioral property which is independent of the implementation details. Figure 2.18 shows an overview of the steps needed to extract these properties for an approximate component. The employed workflow is similar to the methodology used in related works [119, 120].

The FPGA implementation of the component is configurable via generics that are mapped to the entity in the HDL description. These generics are read from a separate configuration file in which they are defined as constants which provides a simple way to move through multiple configurations in an automated process. In order to be able to extract speed and power characteristics which require a pipelined environment, the component is instantiated in a characterization bench between two register stages. For the physical characterization, the component under test is synthesized, placed and routed to the targeted device using the respective vendor-specific tool, e.g. the Intel Quartus Prime Software Suite in case of targeting Intel FPGAs [121]. The area data can then directly be extracted from the vendor tool, which typically lists the consumption of LUTs, registers, BRAMs and DSPs separately. After place & route, the final netlist is also available which is used by the timing analysis tool to calculate all critical path delays in the system which in turn

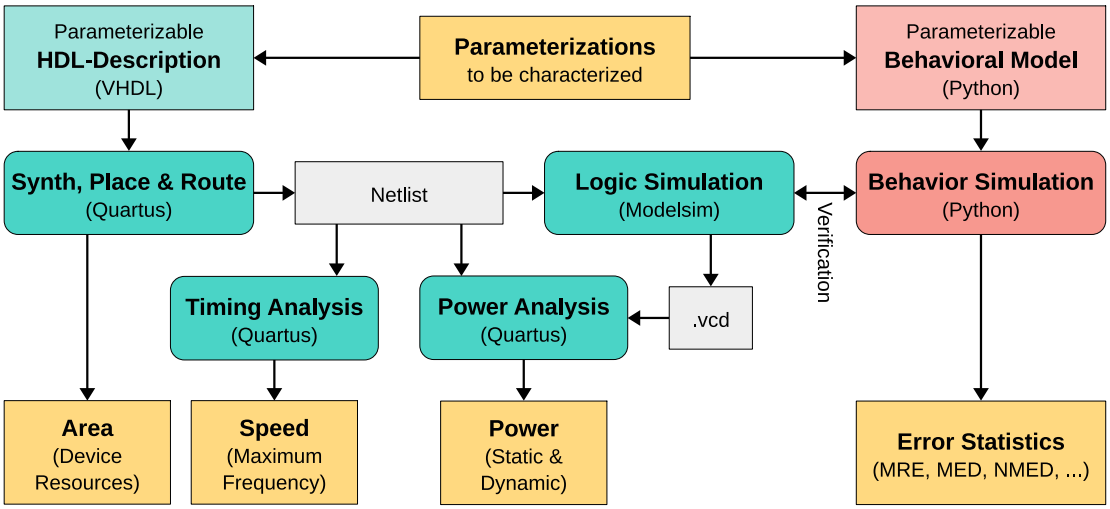


Figure 2.18: Overview of the workflow for the characterization of approximate components

determines the maximum possible operating frequency. The netlist is further used to execute a gate-level accurate logic simulation (e.g. using Modelsim [122]) to provide switching activity data for the power analysis. For this, a simulation testbench is used that feeds the inputs of the components with random input data vectors and generates a `.vcd` file containing the resulting switching data. In this work, we simulated the switching data for a random input sequence of length 100 000. The netlist and the switching data are then used by the respective power analysis tool of the FPGA vendor to extract the power data.

In the place & route step, which maps the synthesized circuit to physical locations on the target device, the process typically starts from a random initial placement and uses a vendor-specific heuristic approach for optimization. When using Intel Quartus Prime, the initial placement can be changed by the designer by configuring a seed value, which will change the result of the place & route step. As the timing and the power data depend on the exact placement and routing, these values change with different seeds. Therefore, to better characterize the average performance of these properties, we repeat the the place & route together will all subsequent steps for 5 different seed values and average the resulting speed and power values.

Regarding the accuracy property, error statistics are calculated via a parameterizable behavioral simulation of the approximate component implemented in Python. These error statistics depend on the input data fed to the component. In a real application, the distribution of the input data largely depends on the application input as well as its internal structure and the position of the respective approximate component within that structure. When characterizing a component independently of any target application, this distribution cannot be known. Therefore, we assume that any input is equally likely to occur in order to estimate the average accuracy loss caused by the approximation in the component. For small input bitwidths, all inputs can be simulated exhaustively but for larger input sizes, this becomes impractical due to computational limitations. Instead, the error behavior of components with larger inputs can be done with Monte-Carlo simulations, using a high number of randomly drawn inputs. In this work, we chose to exhaustively simulate the errors for all components up to input sizes of 26 bits combined across all inputs. For all larger input sizes, we cap the number of simulated inputs at 10^8 values, which are randomly drawn from a uniform distribution.

2.3.2 Statistical Error Metrics

Below, we introduce several arithmetic error metrics which are the most relevant ones for components that produce signals that have a numerical interpretation. For each input to the component, the error distance (ED) is defined as

$$\mathrm{ED} = \left| y_{\mathrm{acc}} - y_{\mathrm{ax}} \right|, \tag{2.23}$$

i.e. the absolute difference between the accurate output y_{acc} and the approximate output y_{ax}. With this, the statistical properties mean error distance (MED) and maximum error distance (MaxED) pertaining to N considered inputs can be calculated as

$$\mathrm{MED} = \frac{1}{N} \sum_{i=0}^{N-1} \left| y_{\mathrm{acc}}(i) - y_{\mathrm{ax}}(i) \right| \tag{2.24}$$

and

$$\mathrm{MaxED} = \max_{0<i<N-1} \left| y_{\mathrm{acc}}(i) - y_{\mathrm{ax}}(i) \right|, \tag{2.25}$$

respectively.

The magnitude of errors naturally scales with the bitwidth of the signal, which makes it hard to compare components of different sizes. To compensate for the component size, the ED can be normalized to obtain the normalized error distance (NED). There are different definitions on what normalization factor to use, with the most prominent ones being the normalization by the maximum output of the accurate design [109] and the normalization relative to the bitwidth of the output [123]. It should be noted that when the maximum accurate output is used as factor, the interpretation of the NED might be problematic for signed output signals because it may take on values larger than 1.0, for example if the accurate output is a large positive number but the approximate component produces a large negative number). To avoid such confusion, we use the second definition and obtain the NED as

$$\mathrm{NED} = \frac{\mathrm{ED}}{2^b}, \tag{2.26}$$

where b is the bitwidth of the component's output.

From that, the normalized mean error distance (NMED) and normalized maximum error distance (NMaxED) can be derived as

$$\mathrm{NMED} = \frac{1}{2^b} \frac{1}{N} \sum_{i=0}^{N-1} \left| y_{\mathrm{acc}}(i) - y_{\mathrm{ax}}(i) \right| \tag{2.27}$$

and

$$\mathrm{NMaxED} = \frac{1}{2^b} \max_{0<i<N-1} \left| y_{\mathrm{acc}}(i) - y_{\mathrm{ax}}(i) \right|, \tag{2.28}$$

respectively. With this definition, the error statistics are given relative to the size of the component which makes it easier to compare components of different sizes.

As an alternative to the ED, the relative error (RE) for an individual input is given as

$$\mathrm{RE} = \frac{\left| y_{\mathrm{acc}} - y_{\mathrm{ax}} \right|}{\left| y_{\mathrm{acc}} \right|}, \tag{2.29}$$

This value gives the error relative to the magnitude of the accurate result and is meaningful when a certain percentage of the accurate result is an acceptable error, i.e. when larger errors can be tolerated for larger outputs. Similar to the definitions above, the mean relative error (MRE) and maximum relative error (MaxRE) related to N inputs can be calculated as

$$\mathrm{MRE} = \frac{1}{N}\sum_{i=0}^{N-1}\frac{\left|y_{\mathrm{acc}}(i) - y_{\mathrm{ax}}(i)\right|}{\left|y_{\mathrm{acc}}(i)\right|} \tag{2.30}$$

and, respectively,

$$\mathrm{MaxRE} = \max_{0<i<N-1}\frac{\left|y_{\mathrm{acc}}(i) - y_{\mathrm{ax}}(i)\right|}{\left|y_{\mathrm{acc}}(i)\right|}. \tag{2.31}$$

Lastly, the bias provides information about the overall shift in the outputs generated across different inputs, i.e. whether the approximated result tends to be larger or smaller than the accurate value on average, and by how much. It is calculated as the average of the actual error in contrast to the MED, which uses the absolute ED:

$$\mathrm{Bias} = \frac{1}{N}\sum_{i=0}^{N-1} y_{\mathrm{acc}}(i) - y_{\mathrm{ax}}(i). \tag{2.32}$$

This value can also be normalized relative to the component size, yielding the normalized bias (NBias):

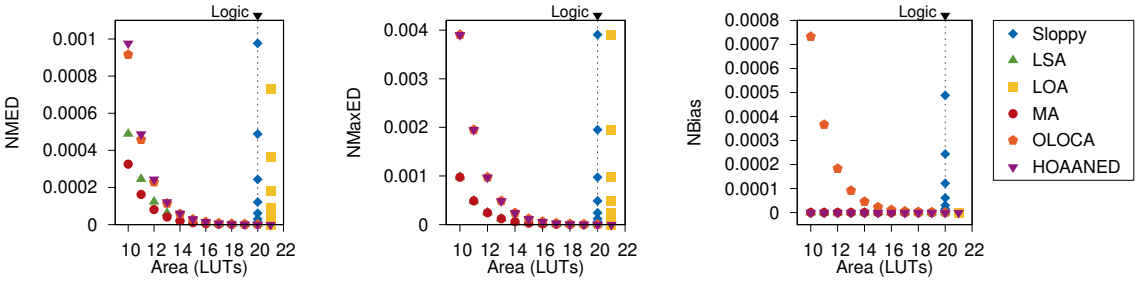
$$\mathrm{NBias} = \frac{1}{2^b}\frac{1}{N}\sum_{i=0}^{N-1} y_{\mathrm{acc}}(i) - y_{\mathrm{ax}}(i). \tag{2.33}$$

Relevance and choice of metrics

The relevance of the metrics introduced above depends on the application domain. For applications that tolerate larger absolute errors when the accurate value is high, the RE and its derived statistical properties are more relevant. However, this might not always be the case. Particularly, in image processing systems, the numerical values associated to the pixels translate into different shades of gray or colors. Here, the same numerical difference is not necessarily less disturbing when the accurate signal assumes a higher value, contrary to what the RE would suggest. In that case, the ED or the NED and their derived statistical properties are more meaningful. However, it should be noted that while these signal difference metrics provide a good way to compare the performance of competing components, they may not be suitable for judging the output quality of a complex application. For that purpose, application-specific quality metrics often give better and more interpretable results and should therefore be preferred for quality judgment on the application level.

2.3.3 Comparison of Approximate Adders

We characterized all implemented adders using the proposed procedure for several input sizes. For the error estimation, we assumed that the inputs are pre-pended each by a single `0`, i.e. that only inputs $0 \leq a, b < 2^{n-1}$ are fed into an n bit adder, which is usual practice to avoid overflow. In terms of error metrics, the results exhibit similar trends for the NED and the RE, so we discuss the NED-related metrics only as well as the Bias. Figure 2.19 plots the NMED, the NMaxED and the Bias as function of area in terms of utilized LUTs for 20b adders.



(a) NMED over Area (b) NMaxED over Area (c) NBias over Area

Figure 2.19: Error metrics over area for 20b adders (the area consumption of accurate logic indicated as vertical line)

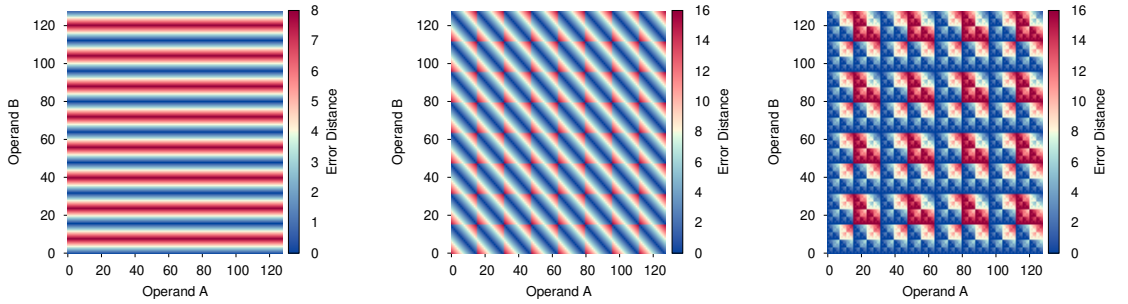
As expected, the accurate logic implementation consumes 20 LUTs. The Sloppy Adders also compute all output bits logically and hence also consume 20 LUTs whereas the LOAs need one additional LUT to generate the carry-in for the accurate part. Comparing these two adders at the same carry-split configuration, the LOA lowers the NMaxED by almost 50% and the NMED by roughly 25% at the cost of the additional LUT. In terms of resource usage, these two adders do not yield any benefits besides the speed improvement enabled by the carry split. However, the adders employed in the target application scope of low-level in-camera image processing are fairly small. For example, all adders employed in the case studies presented in this thesis are smaller than 32 bits. However, the slowest 32b adder we characterized operates at up to 561.15 MHz while the fastest ones are restricted by the maximum frequency supported by the target FPGA device, which is 645.16 MHz. Hence, all adders within our scope are sufficiently fast and the speed improvements generated by the carry chains are only relevant for large adders or when many additions are chained between pipeline registers.

In contrast to the Sloppy Adder and the LOA, the area consumption of the other adders scales with the split point. Considering the area-NMED trade-off, the MA performs best, followed by the LSA. Both of these adders remove the computation completely below the split point. The OLOCA as well as the HOAANED perform significantly worse in this trade-off, with the OLOCA showing slightly better results than the HOAANED. When using the same carry-split point, these two have lower errors than the former two, but they consume more logic since both of them still compute the two upmost bits in the approximate part.

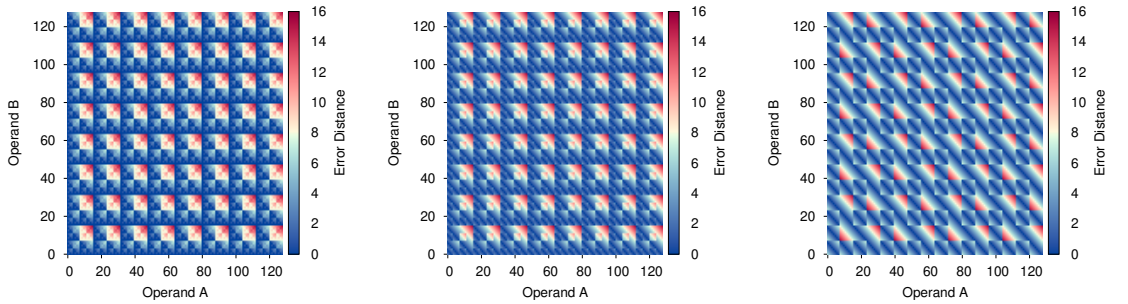
In terms of the NMaxED, a similar trend is visible, but the differences between some of the competing adders get smaller. At the same area consumption, the MA and the LSA show exactly the same NMaxED. Similarly, the OLOCA and the HOAANED have almost the same maximum error. On the other hand, looking at the bias, the LSA, the LOA, the MA and the HOAANED perform similarly well and barely produce an offset in the average output. In contrast, the Sloppy Adder

and the OLOCA exhibit increasing bias when the split point is raised, with the OLOCA providing a trade-off between area and bias while the area of the Sloppy Adder is fixed. However, all adders have a positive bias, which means that they tend to produce smaller results than the accurate one.

It should be noted that all error metrics used for this discussion assume that the inputs to the units are uniformly distributed. When specific patterns or regularities are present in the data that is fed to them in an actual application, their trade-off performance may vary. The error behavior of different approximate units across the input values can be observed in the error maps, which are plotted exemplary for 8-bit adders with similar MED values in Figure 2.20.



(a) LSA (split=4, sel=A, MED=4.00) (b) MA (split=4, MED=5.31) (c) LOA (split=4, MED=5.88)



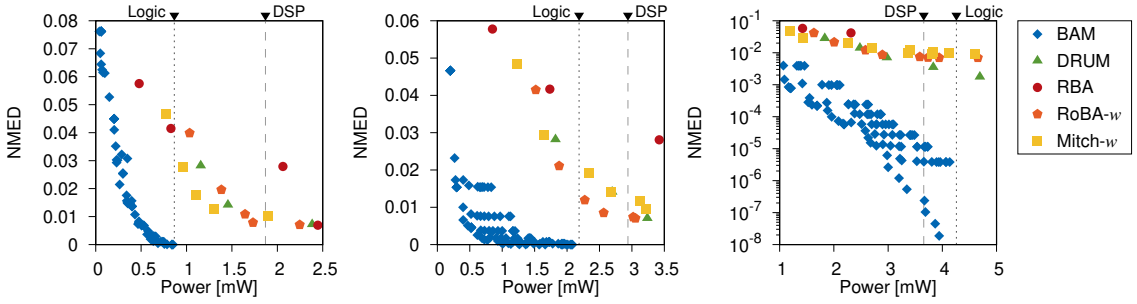
(d) Sloppy Adder (split=4, MED=3.75) (e) OLOCA (split=4 ,MED=3.70) (f) HOAANED (split=5 ,MED=3.97)

Figure 2.20: Error maps of 8-bit adders (fed by 7-bit inputs pre-pended with a `0` to allow for overflow)

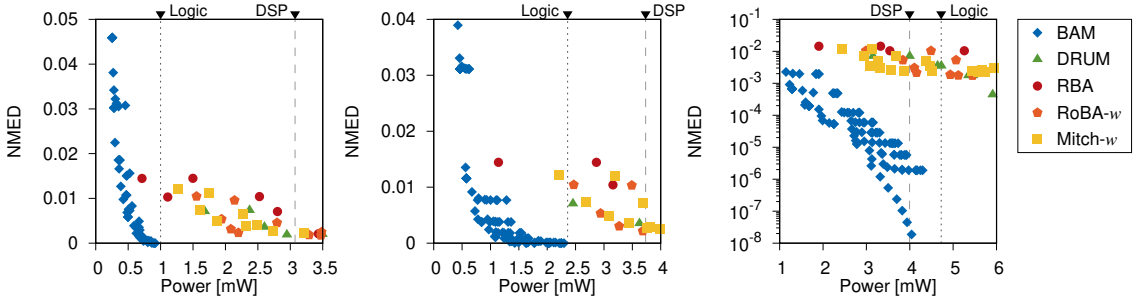
The plot shows that the distribution of error values follows different regular patterns. The LSA behavior differs from the other adders in that the error pattern depends on the input selected for pass-though in the approximate part while the other adders behave symmetrically for both inputs. Depending on the actual distribution of input values, this could be a benefit or a drawback, e.g. when one of the input changes significantly more than the other one, the LSA could be configured to pass through the input that changes. Furthermore, the LSA has a smaller MaxED compared to the other ones at a similar MED value. For all other adders, the error map structure is very similar, with smaller sub-patterns repeating along both dimensions. In contrast to the LSA, they are agnostic to the input order, i.e. they behave the same when the inputs are interchanged. Structurally, the OLOCA is a mixture between the MA (for the lower bits) and the Sloppy Adder (for the two top-most approximate bits), which is reflected in the respective error map. Additional similarities can be found between the MA and the HOAANED due to the similarity of the lowest bits in the approximate part.

2.3.4 Comparison of Approximate Multipliers

Similarly to the adders, we characterized the selected multipliers across several sizes. First, we discuss the general power-error trade-off across several multiplier sizes. Figure 2.21 plots the power-NMED trade-off of unsigned and signed approximations of 8×8, 10×15 and 16×16 multipliers, spanning multiple overall sizes as well as equal and unequally sized inputs. The plot also shows the consumption of the accurate logic implementation and the DSP multiplier. For smaller units, the logic implementation is more power efficient than the DSP but this trend is reversed for the 16×16 size.



(a) Unsigned 8×8 (b) Unsigned 10×15 (c) Unsigned 16×16



(d) Signed 8×8 (e) Signed 10×15 (f) Signed 16×16

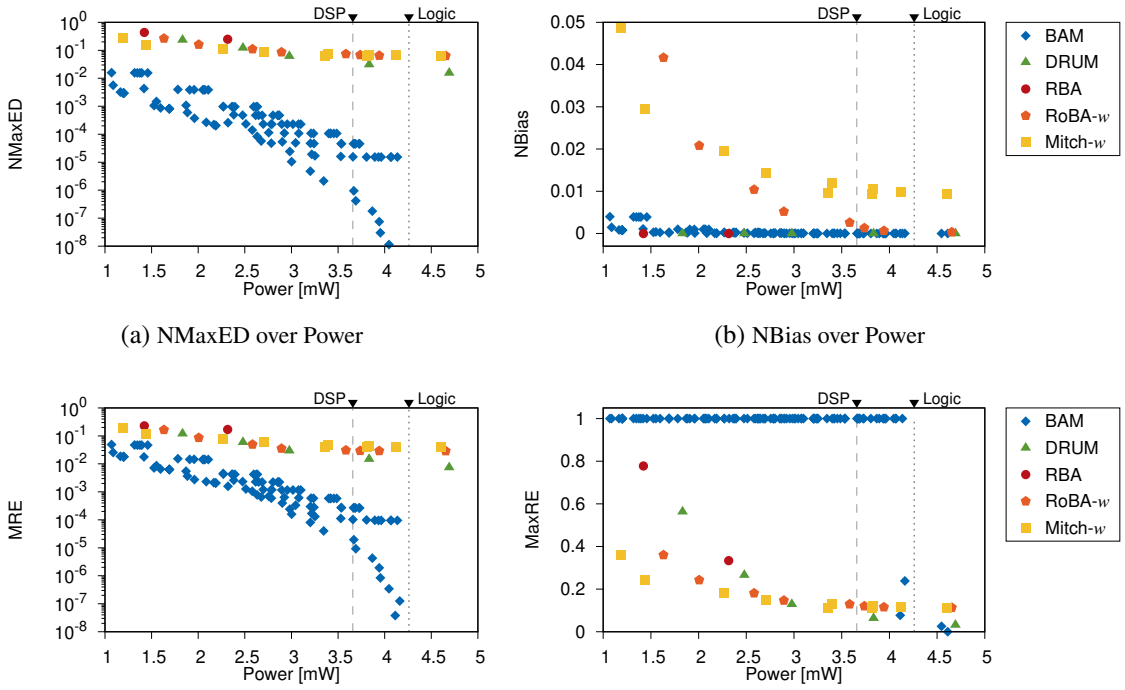
Figure 2.21: NMED over power for unsigned/signed 8×8, 10×15 and 16×16 multipliers (the power consumption of the accurate logic and DSP implementations are indicated by vertical lines; plots (c) and (f) use a logarithmic scale at the error dimension for better visibility)

The plots show that at all sizes and signedness choices, the approximate multipliers offer a range of trade-off options to reduce the power consumption at the cost of some error. This trade-off space is generally dominated by the BAM which outperforms the competing options across all configurations. It seems that the other adders are held back by some general overhead because especially for smaller sizes, many of their configurations consume more power than the accurate versions. Furthermore, the BAM configuration offers two parameters, HBL and VBL, which allow for a finer tuning of the trade-off compared to the others. Regarding the signed variants, the DRUM, the RBA, the RoBA-w and the Mitch-w use a sign conversion wrapper and offer a choice between approximate and accurate sign conversion. The characterization data shows that in all cases the accurate sign conversion introduces a significant overhead while reducing the error only by a small

amount. In the plots, this is reflected in pairs of these multipliers appearing almost along horizontal lines. Generally, the approximate sign conversion should be preferred in terms of the trade-off.

Figure 2.22 plots different error metrics over the power consumption for unsigned 16×16 multipliers for further analysis. First, the plot in Figure 2.22a shows that the behavior regarding the NMaxED is similar as for the NMED. The second plot (Figure 2.22b) depicts the behavior of the normalized bias. It shows that all DRUM variants exhibit almost no bias, which is one of its main design goals. Similarly, the two relevant RBA variants and most of the BAM multipliers have a very small bias. The RoBA-w variants trend towards larger bias at higher degrees of approximation but approach zero bias at higher quality settings whereas the Mitch-w variants show a considerable amount of bias even at high quality settings. As observed above for the approximate adders, the bias of all multipliers tends to be positive, i.e. the approximate output is generally smaller than the accurate one.

Finally, Figures 2.22c and 2.22d show the trends for MRE and MaxRE, respectively. While the power-MRE trade-off behaves similar to the NMED and NMaxED counterparts, the MaxRE follows a different trend. For this metric, almost all BAM variants have a MaxRE of 100% while the other approximate multipliers are largely constrained between 10% and 40% of the accurate output. This is largely due to the fact that the BAM variants may produce a zero output for small inputs if the corresponding rows and columns in the partial product array are completely removed, while the error magnitude scales better with the input values for the other units (see also the error maps discussed below).



(a) NMaxED over Power

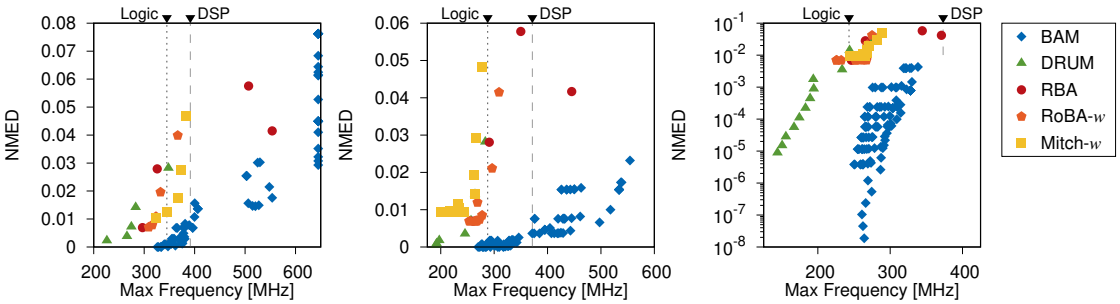
(b) NBias over Power

(c) MRE over Power

(d) MaxRE over Power

Figure 2.22: Error metrics over power for 16×16 multipliers (the power consumption of the accurate logic and DSP implementations are indicated by vertical lines; plots (a) and (c) use a logarithmic scale for the error dimension for better visibility)

Next, we analyze the speed characteristics of the different multipliers. To that end, Figure 2.23 plots the NMED over the achievable speed when placed between two pipeline register stages in terms of MHz. Note that generally for the target FPGA device, the overall maximum frequency is limited to 645.16 MHz, which is consequentially the upper limit for the multiplier speed as well. The plots show that similar to the power characteristics, the BAM variants perform best when implemented for the target FPGA device. Furthermore, it can be seen that as the size increases, the performance gap between the DSP multiplier and the accurate logic implementation increases.

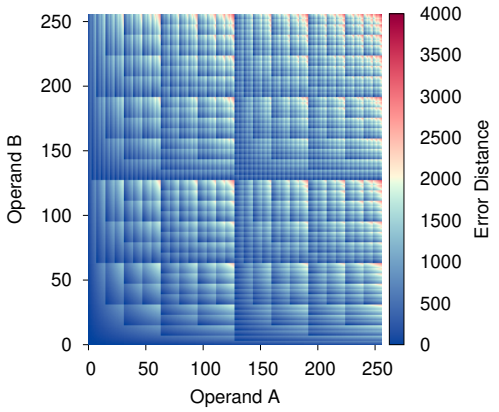


(a) 8×8 (b) 10×15 (c) 16×16

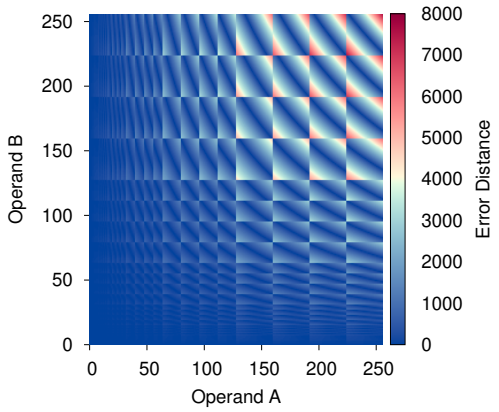
Figure 2.23: NMED over speed for unsigned 8×8, 10×15 and 16×16 multipliers

Lastly, we plot exemplary error maps of the different multiplier types in Figure 2.24. Two general trends can be found among the different methods. For the BAM, the errors are distributed roughly uniformly across the entire input space, with a few peaks towards higher values. For all other types, in contrast, the error magnitude scales with the input magnitude, forming areas of similar error levels with exponentially increasing boundary intervals in both input operand dimensions. The average as well as the peak error aggregated in individual error plateaus also rise exponentially. While the the RBA (in operation mode 3), the RoBA-w and the Mitch-w concentrate the highest error values towards the middle of the plateaus, the DRUM spreads the errors more evenly within these areas, which is due to the unbiasing step in its implementation. Overall, the error distribution of the BAM favors uniformly distributed inputs while the other alternatives perform better with input distributions that are skewed towards smaller values, and when relative error performance is more relevant than absolute errors.

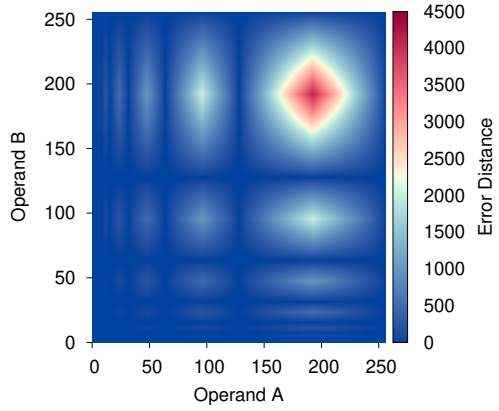
To summarize, this analysis shows that the BAM generally translates best to FPGA architectures, across all tested sizes. It outperforms the other multipliers in terms of resource efficiency as well as speed performance and generally has lower errors in all metrics except for the MaxRE. In contrast to the BAM, the other methods improve their performance with increasing overall multiplier size. This might be due to the fact, that unlike the BAM which simply removes parts of the partial product array, they require a custom structure, causing some implementation overhead especially at smaller sizes. It is possible that these methods might outperform the BAM in sizes even larger than the tested ones, but similar to the adders, the multiplier sizes found in the application scope targeted in this thesis are generally constrained within the compared range. Furthermore, at larger sizes, the DSPs become more efficient and might be hard to beat in terms of resource efficiency and speed. In conclusion, the BAM should be the preferred choice among the selected multipliers in nearly all situations except for very specific exceptions, e.g. when the MaxRE performance is relevant or when the input is known to follow specific patterns.



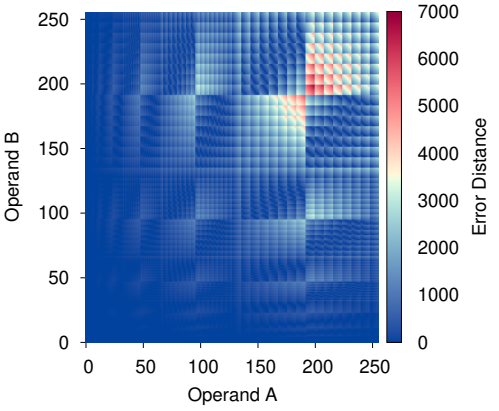
(a) BAM (HBL=2, VBL=9), MED=896



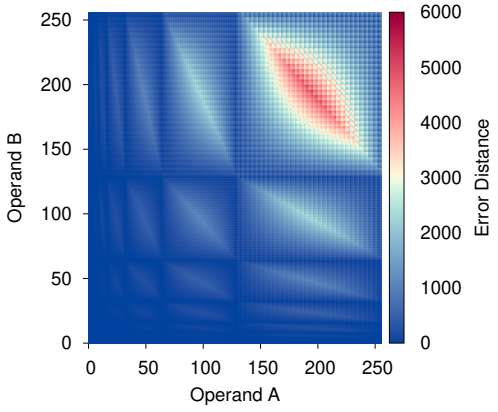
(b) DRUM (k=4), MED=930



(c) RBA (mode=3), MED=455



(d) RoBA-w (w=5), MED=712



(e) Mitch-w (w=6), MED=828

Figure 2.24: Error maps of 8×8 multipliers

2.4 Modeling Component Characteristics

To model hardware characteristics of entire applications, our proposed framework uses a divide-and-conquer strategy that composes the area consumption as the sum across all individual system components and derives the power consumption from the number of employed FPGA resource units as described in detail in Section 3.3. This means that the framework needs to be able to assess the resource usage of individual components from data in the component library. While for some of the components, the resource usage can be calculated accurately directly from their parameterization, others need to be characterized by synthesis. For the components selected and implemented for this work, this especially holds for the approximate multipliers (see Section 2.2.3) and the premapper block that calculates the address and interpolation offset for the selected sparse table method (see Section 2.2.4). In both of these cases, a combinatorial block is generated according to the parameterization whose resource consumption is affected by the synthesis optimization and fitter placement in the FPGA vendor toolchain. However, the design space formed across the possible component sizes and approximation parameters of these components is too large to be synthesized entirely for an exhaustive characterization. As an example, considering all possible sizes between 4×4 and 18×18, the BAM multiplier supports 14 540 potential configurations each for unsigned and signed operation.

To overcome this limitation, we employed supervised machine learning (ML) algorithms to train component models using data obtained from the characterization of a sampled subset of configurations. Our approach uses a workflow similar to *ApproxFPGAs*, in which ML models are used to predict FPGA-specific characteristics of approximate circuits that were originally designed for ASIC implementations, using their respective hardware description as model inputs [124]. However, their models are specific to a few fixed component sizes, namely 8-bit, 12-bit and 16-bit arithmetic units. In our case, the component characteristics need to be predicted across a wide range of component sizes, taking the size itself together with the internal approximation parameters as input to the models. An overview of the entire model formation process is given in Figure 2.25.

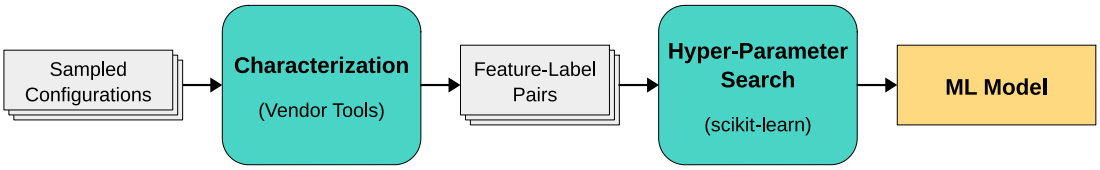


Figure 2.25: Overview of the ML model formation procedure

After the characterization, the data set consists of feature-label pairs, where the feature vector of each pair describes the component configuration and the corresponding label holds the value to be modeled, e.g. the number of LUTs consumed by the component. The whole data set is split into a training set containing 80% of all available data that is used to obtain the final model and a validation set (20%) which is used to evaluate the accuracy of the final model on new data excluded from the training. We used the Python toolbox `scikit-learn`[1] [125] to train *multi-layer perceptron (MLP)* models, i.e. feed-forward artificial neural networks with one or more hidden layers, as well as *random forest (RF)* models, which consist of a collection of de-correlated decision

[1] `https://scikit-learn.org`

trees whose predictions are averaged [126]. These two model types where chosen because they performed best among the regressor models available in the toolbox in initial experiments.

To improve the model accuracy, we performed a randomized hyper-parameter search for both model types. In case of the MLP, we optimized the *network topology*, i.e. the number and sizes of the hidden layers, the *regularization coefficient*, which counteracts overfitting by penalizing large node weights, and the *learning rate*, which controls the step size in which the weights are adapted during the training [127]. On the other hand, for the RF models, we optimized the *number of estimators*, i.e. the number of employed decision trees, the *maximum tree depth*, the minimum number of samples to split an internal node (*min_samples_split*) and the minimum number of samples required per tree leaf (*min_samples_leaf*) [128]. During the hyper-parameter optimization, a k-fold cross-validation strategy is used for choosing the best parameter set [129]. For this, the training data is partitioned into k distinct folds. Then, for each of the tested hyper-parameter combinations, k different models are learned, each using $k-1$ of the folds for training and the remaining fold for independent validation, and the performance is given by the average validation score across all k iterations. For our model, we use 10-fold cross-validation. Specific information on the trained models and the obtained accuracies for the multipliers as well as sparse table premapper blocks are given in the subsequent sections.

2.4.1 Approximate Multiplier Models

The approximate multipliers are configured by two types of parameters: their size, which depends on the width of the input signals, and the internal approximation parameters. Without loss of generality, we restrict the target size range between 4×4 and 18×18 multipliers, which covers typical ranges in the target application scope and accommodates the case studies presented later in this work. In this range, there are 120 possible sizes when inverted operand pairs, i.e. $A\times B$ and $B\times A$ are counted only once. From those, we deliberately had selected 7 different sizes, namely 4×4, 8×8, 12×12, 16×16, 4×12, 5×15 and 10×15 and additionally 21×21, which is slightly outside the range, to manually study the characteristic behavior of different multiplier types. The resulting data can be re-used for model training. Additionally, we randomly chose 44 additional sizes from the range to obtain further training data. For each size, we sampled multiple approximation parameter settings across the internal design space. To take the resulting structure of the training data into account during the model selection and validation process, we grouped the samples according to the multiplier size and forced the training/validation data split as well as the k-fold split to be between groups. This ensures that the multiplier sizes of samples used for model validation purposes have not been seen during the respective training and hyper-parameter optimization steps.

Area Models We trained separate area models for each combination of multiplier type and signedness. For the DRUM, the RBA, the RoBA-w and the Mitch-w, we only considered the variants with approximate sign conversion (which is the preferred option for signed operation as shown in Section 2.3.4). As described above, hyper-parameter optimizations were performed for each model to find the best MLP and RF parameters and the cross-validation score of the respective best model was used to select between MLP and RF.

Table 2.4 lists the chosen model type (MLP or RF) for each of the trained models as well as its accuracy in terms of average absolute (MED) and relative (MRE) prediction error, computed over the held-out validation set. The MLP was selected for all models as it consistently outperformed

the RF. The final models are able to estimate the area consumption of previously unseen data with an average absolute deviation between 2 and 12 LUTs and an average relative deviation between 4 and 9 percent, with the unsigned BAM and both Mitch-w models showing the highest relative deviation. In terms of absolute errors, the BAM models have the highest accuracy, and the DRUM as well as the Mitch-w perform worst. Overall, the accuracy of all models is appropriately high to be used within the framework for fast resource estimation without time-consuming synthesis.

Table 2.4: Accuracy of approximate multipliers area models (modeling LUT usage)

Multiplier Type	Signedness	Model Type	MED	MRE
BAM	unsigned	MLP	2.66	8.26
	signed	MLP	2.75	4.57
DRUM	unsigned	MLP	5.47	3.43
	signed	MLP	9.89	5.09
RBA	unsigned	MLP	4.37	4.74
	signed	MLP	4.83	4.58
RoBA-w	unsigned	MLP	5.28	3.96
	signed	MLP	5.38	3.82
Mitch-w	unsigned	MLP	9.63	8.11
	signed	MLP	11.24	8.38

Speed Models This work focuses on reducing the resource usage at a given operating frequency instead of increasing the speed of the application. However, when selecting and parameterizing components for use within an application, it needs to be ensured that the approximated system does not violate any timing constraints. Generally, the framework maintains the structure and placement of register stages of the target application and assumes that the approximation-less reference system can safely operate at the target speed. Then, the selection process only needs to ensure that no new timing violations are introduced with the approximations. We found that the selected multipliers are time-critical at target frequencies between 200 and 300 MHz, as shown in Section 2.3.4.

For this work, we assume that multipliers are to be placed directly between register stages, which holds for all presented case studies. If this assumption breaks, further measures need to be implemented. To ensure safe selection of approximate multipliers at a given frequency, we train additional speed models which can be used together with a safety margin to restrict the choice of multiplier types and parameters. The selected model type and the respective accuracies of the final model obtained for the validation set are shown in Table 2.5.

The data shows that the models capture the behavior of DRUM and RoBA-w best, while for BAM, the average deviation is roughly 17 MHz. However, the BAM generally reaches the highest speeds across all selected multipliers. In terms of average relative error, the deviation lies between 1% and 7%, depending on the multiplier type. Unfortunately, it cannot be known exactly how well even the characterized estimation of multiplier speed holds up when embedded in larger systems, because the synthesis and fit/route steps are influenced by many factors. Nevertheless, these numbers enable setting a safety margin above the target system speed and filtering out the components for which the respective model predicts a threshold violation to reduce the risk of timing violations.

Table 2.5: Accuracy of approximate multipliers speed models (modeling maximum frequency in MHz)

Multiplier Type	Signedness	Model Type	MED	MRE
BAM	unsigned	RF	17.09	4.16
	signed	RF	17.06	4.60
DRUM	unsigned	RF	3.50	1.42
	signed	RF	3.96	1.81
RBA	unsigned	RF	11.84	2.88
	signed	RF	11.25	3.07
RoBA-w	unsigned	RF	3.93	1.41
	signed	RF	4.36	1.76
Mitch-w	unsigned	MLP	14.78	6.39
	signed	RF	12.63	5.52

2.4.2 Sparse-Table Premapper Models

The premapper blocks of the hierarchically segmented sparse tables as introduced in Section 2.2.4 are parameterized by several main parameters. First, the interpolation flag decides whether the offset of the input from the selected segment needs to be calculated for interpolation, which requires additional logic. Secondly, the segmentation is defined by the number of sections and a list containing the number of segments within each section. We trained individual area models for each combination of the interpolation flag and the number of sections. When only 2 sections are used, the premapper design space contains only 144 different configurations, which can easily be characterized exhaustively. However, with more sections, the number of possible configurations rises significantly, yielding approximately

- 1.46×10^4 combinations with 4 sections,
- 1.00×10^8 combinations with 8 sections,
- 1.85×10^{15} combinations with 16 sections and
- 7.92×10^{28} combinations with 32 sections.

For model creation, we characterized 1000 random parameterizations for the variant with 4 sections and 10000 for the variants with 8, 16 and 32 sections to form the respective data sets. Table 2.6 shows the chosen model types and validation set accuracies for each of the final models.

For all non-exhaustive cases, the RF models outperformed the MLP models. It can be seen that with increasing diversity of the design spaces, the prediction accuracy becomes worse, from average errors below 1 LUT respectively below 5% for the models with 4 sections up to average deviations below 13 LUTs respectively below 12% for the variants with 32 sections, which are generally more complex. However, especially for the latter case, the created data set only covers $1.26 \times 10^{-23}\%$ of the design space. Hence, the accuracy could probably be improved by characterizing more configurations at the cost of significantly increased time effort if increased accuracy would be necessary.

Table 2.6: Accuracy of sparse table premapper area model (modeling LUT usage)

Multiplier Type	Interpolation	Model Type	MED	MRE
2	□	Exhaustive		
	■	Exhaustive		
4	□	RF	0.49	4.64
	■	RF	0.74	4.49
8	□	RF	1.93	9.24
	■	RF	2.06	7.12
16	□	RF	3.69	10.10
	■	RF	4.88	8.02
32	□	RF	8.67	11.60
	■	RF	12.81	9.75

2.5 Library of Approximate Components

The approximate components that were selected and implemented are stored in a library from which they are available to be used in the DFG of a target application as explained later in Section 3.1. Each component in the library comprises three major elements: a parameterizable hardware implementation, a parameterizable behavioral model and a resource model. This section provides details on the implementation and the interfaces of these elements as well as a summarized overview of the components included in the library.

2.5.1 Implementation and Interfaces

On the hardware side, each component is implemented as an entity in `VHDL` that is parameterizable in I/O width and approximation strength via generics. For the arithmetic units, where multiple competing methods respectively their implementations exist, the top-level entity of the component is a generic wrapper that instantiates the selected implementation. Since the precision scaling only influences the bitwidth of internal signals, it is implemented on the application level, where generics control the widths of internal signals. To form the hardware implementation of an entire application, the approximate components are instantiated inside a top-level entity which is parameterized using a list of generics that are forwarded to the respective component interfaces.

The component models, on the other hand, are implemented in object-oriented `Python`, which is also used for the further parts of the framework presented in the subsequent chapters. Each component type is represented as a class that provides a `process()` method to access the behavioral model as well as a `report_area()` method for the resource model. The behavioral model is a fast, bit-exact implementation of the operation carried out by the component while the resource model provides an estimate of the consumed FPGA resources, which is either derived directly from the parameterization, if possible, or accesses the respective ML model trained from characterization data as described in Section 2.4. The parameterization of individual components is stored in the properties of any specific object instance so that it can be taken into account by the models.

2.5.2 Library Contents

The library currently contains all the approximate components selected and implemented in Section 2.2. It can easily be extended in the future by adding more implementations that adhere to the interfaces described above. Table 2.7 summarizes the approximate components and methods currently implemented and available to be used in the framework, listing the configuration parameters as well as the type of the resource model for each component.

Table 2.7: Overview of approximate components/methods currently available in the library

Type	Method	Parameters	Resource Model
Precision Scaling	Adapting internal signal widths	Width	Analytical
Adders	Accurate	Width	Analytical
	LSA	Width, Split, Input Select	Analytical
	MA	Width, Split	
	LOA	Width, Split	
	Sloppy Adder	Width, Split	
	OLOCA	Width, Split	
	HOAANED	Width, Split	
Multipliers	Accurate	Width A & B	Exhaustive Characterization
	BAM	Width A & B, HBL, VBL	MLP / RF
	DRUM	Width A & B, Core Size	
	RBA	Width A & B, Operation Mode	
	RoBA-w	Width A & B, Truncation Width	
	Mitch-w	Width A & B, Truncation Width	
Function Table	Hierarchically Segmented Sparse Table	No. of Sections, No. of Sub-segments	Premapper: Exhaustive / RF Memory: Analytical

2.6 Chapter Summary

This chapter has bridged the gap between the publication of approximation methods and their practical use within FPGA-based systems. It provided a thorough and structured review of the related literature based on which suitable methods have been selected under consideration of the target application scope. Various approximate arithmetic units and a hierarchical table-based approximation of elementary functions have been implemented for use as approximate system components. The implementations are flexibly scalable to allow a purposeful combination with fine-grain precision scaling. Furthermore, the individual components were characterized in terms of hardware and error properties, and the quality-resource trade-off was compared for competing adders and multipliers. In addition, the characterization data was used to train ML models for a fast estimation of hardware properties. Finally, all implementations and models were combined into a library of approximate components with common interfaces to facilitate their flexible deployment.

Chapter 3

Application Modeling

Real-world signal processing applications use a sequence of operations to process the incoming data. In hardware-based computing systems, these operations are performed by individual components, for example arithmetic units, which are connected via signals. When integrating approximations into such a system, the standard implementation of any component may be replaced by an alternative approximate version which reduces the resource consumption of the component at the cost of introducing errors in the operation. Additionally, precision scaling can be used to directly adapt the width of the connecting signals. A combination of multiple approximations applied at different points across the application may be employed to optimally exploit the potential quality-resource trade-off. However, specific properties of approximated components or the direct application of precision scaling can impact the width of intermediate signals, leading to dependencies between the parameters of connected components which need to be managed. These factors result in design spaces that are both large and complex and an automated DSE becomes necessary to find optimal parameterizations.

During the DSE, many candidate solutions are probed and their fitness in terms of resource usage and application quality needs to be evaluated. However, traditional means of assessing the resource usage of an application, i.e. performing synthesis, fitting and routing, are very time-consuming. Similarly, performing exhaustive gate level simulations for quality assessment would be very costly. Consequentially, suitable models for a fast and accurate estimation of the overall application-level resource usage and output quality are needed to accelerate the fitness estimation during the DSE. Furthermore, such models need to account for parameter interactions between individual components as well as error propagation effects across the application.

To accommodate these requirements, this chapter proposes methods to manage the parameterization of a target application that combines different approximation types and presents models to accelerate the quality-resource estimation. First, Section 3.1 describes the representation of a target application in form of an *annotated DFG* that captures the associated design space and is used as basis for the resource and quality models. Then, Section 3.2 explains the causes for and impacts of parameter dependencies between system components and proposes methods of handling them. Building upon the DFG-based representation of the application, the proposed resource and quality models are presented in the remaining sections. Section 3.3 proposes a divide-and-conquer approach to model the overall number of consumed device resources, from which the power consumption is then derived. Finally, Section 3.4 presents the proposed quality model which allows the designer to flexibly employ their preferred application-specific reference metric and discusses the choice of relevant training data sets for the quality estimation.

3.1 Annotated Data Flow Graph of the Application

The proposed framework uses a DFG to represent the structure of a target application. Each node in the graph represents a component or operation in the system and the graph's edges are directed and represent the data flow between components. In practical terms, the DFG is formed by the application designer by writing a function in which the nodes and connections are created, using a standard graph library called `networkx`[1] [130]. To define the functionality of each node, an instance of the respective component class taken from the component library (cf. Section 2.5) is created and stored with the node. These instances also store information about the placement of register stages in the component output, which needs to be provided by the designer.

The components in the DFG that support approximation can be parameterized by setting the respective object properties to switch between normal or approximated operation, to select the specific approximation method and to tune the degree of approximation. When creating the DFG, the designer can enable the use of approximations by individually marking suitable components and setting ranges for the associated parameters. The resulting DFG that contains these notations will further be denoted as *annotated DFG*. This nomenclature is inspired by state-of-the-art methodology for approximated software where variables and operations are annotated to mark their suitability for approximation [131, 132]. Figure 3.1 provides an exemplary illustration of the annotations in the DFG of a simple channel mixer application. This circuit takes three input signals representing different color channels of an image and calculates a weighted sum according to the coefficients $m_{0,1,2}$. Its approximate version uses a combination of precision scaling and approximate arithmetic units. Note that the final shift node is not configurable because it is used to shift the result to a fixed bitwidth required by the output.

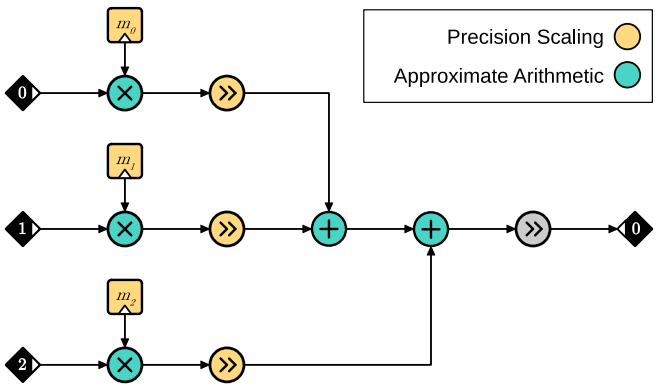


Figure 3.1: Illustration of an annotated DFG for a simple channel mixer. Components marked for different types of approximation are color-coded. Associated annotated parameter ranges are not shown.

Candidate DFGs The annotated DFG represents the entire design space of the approximated application, which needs to be explored in order to exploit potential quality-resource trade-offs. It yields a list of all parameters that control the overall configuration of the application together with the specified ranges. In contrast, any fixed configuration within the design space will be denoted as *candidate DFG*, in which the parameter properties of all component instances are set to specific

[1] `https://networkx.org`

values. Hence, the candidate DFG represents a specific solution for the DSE problem associated with the annotated DFG. During the fitness evaluation of a candidate solution, the candidate DFG is used by the resource and quality models as described in Sections 3.3 and 3.4, respectively.

Behavioral DFG Simulation The framework provides functionality to simulate the approximate application output related to any candidate DFG. For this, the designer needs to mark specific DFG components as input and output nodes, and associate a color channel to them. The input image is then split into the different color channels and the data of each channel is stored with the input nodes. Then, a recursive procedure traverses the DFG to calculate the output signals of all components based on the outputs of the respective preceding components using the individual `process()` methods of the parameterized component instances. Finally, the output image can be reconstructed from the data stored with the output nodes. Since the `process()` methods themselves provide a parameterization-dependent, bit-accurate simulation of the components, the DFG simulation implicitly accounts for error propagation effects.

Current Limitations As mentioned in Chapter 1, this dissertation targets image stream processing pipelines that take a sequence of single pixels and process them. The presented case studies focus on color processing applications. In its current state, the framework does not support applications that employ spatial processing, i.e. where multiple pixels in a local neighborhood are involved to produce the output pixel. Examples for this would be finite impulse response (FIR) filters, noise reduction or motion estimation. Appropriate extensions to support such systems are conceivable and could be part of future work, as discussed in Chapter 6.

3.2 Parameter Dependencies

Depending on the structure of the application and the composition of potential approximations, the configuration of the DFG might be subject to dependencies between the parameters of individual components. In order to ensure the generation of valid and synthesizable candidates during the DSE phase, these dependencies must be respected. Additionally, specific characteristics in the input or output of approximated components might enable the synthesis tool to perform further optimizations across multiple components, reducing the resource usage of the system. Hence, such interactions must also be considered to achieve an accurate estimate of the real resource count for any candidate DFG. In the following, we consider the color channel mixer DFG introduced above to illustrate cross-component parameter dependencies and explain how they are handled in our system.

3.2.1 Parameterization Order

An exemplary parameter propagation path in the mixer might start in the node holding coefficient m_1. When precision scaling is applied here, changing the output bitwidth of the node, the size of the subsequent multiplier will change which in turn influences its available approximation parameter range. Next, the size of the multiplier output, which is determined by the sizes of its inputs together with its configuration, defines the range of the bit-shift operation that follows. The path continues as the output of the bit-shift, together with the output of the bit-shift in the second channel, defines the size and parameter range of the first adder, and so on. Generally, the parameter ranges

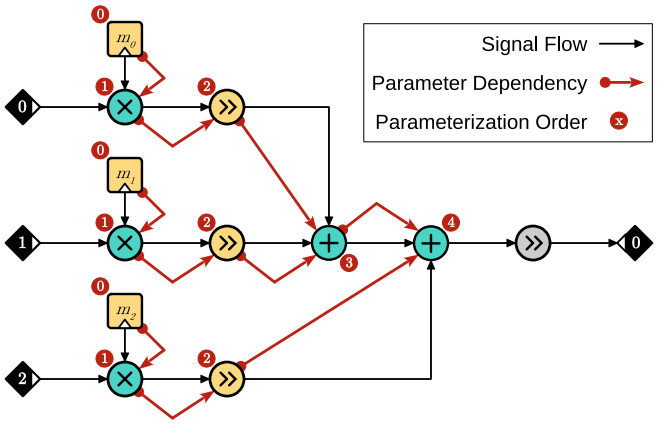


Figure 3.2: Forward propagation of legal parameter ranges and derived order of parameterization

of individual DFG components typically depend on the size of their respective inputs, leading to a forward propagation of parameter dependencies. Figure 3.2 illustrates these propagation paths in the exemplary channel mixer DFG.

Consequentially, the parameterization should follow the same order during the generation of new or modified candidate solutions. For this, we group all components into parameter groups according to the level of parameterization dependency relative to preceding components. First, all independently parameterizable components are assigned to group 0. Then, all remaining components are iteratively sorted into groups with higher indices so that the members of any group solely depend on components in lower groups. The group index then defines the order of parameterization during the formation of new candidates, as indicated with numbers in Figure 3.2.

3.2.2 Synthesis Optimizations

Another form of cross-component interaction can be observed during the synthesis step of a candidate system, where synthesis optimizations lead to reductions in resource usage. This happens when components either produce some constant bits in part of their output or when they disregard some bits from their inputs. In the channel mixer example, the approximation used in any of the multipliers might produce constant `0`s in some of the LSBs of its output, for example when employing the BAM. However, the synthesis tool will automatically remove any register fed by a constant signal. Also, if some bits in one of the inputs to an adder are constantly set to `0`, the respective adder logic can be removed and the other input can be promoted directly to the output. This behavior reflects a forward propagation of synthesis effects.

However, the parameterization of a component might as well influence the preceding components. As an example, the final adder in the channel mixer could implement the LSA approximation, which selects bits from one of its two inputs in the lower part and disregards the respective bits from the other input. Generally, some of the input bits to a DFG component might be disregarded, which effectively reduces the size of that signal. Consequentially, these bits do not need to be calculated by previous nodes. Such cases lead to a backward propagation of synthesis effects.

In both cases described above, the DFG configuration is still valid, but the synthesis tool will remove unnecessary resources in the related components. In order to avoid an overestimation of the overall resource consumption, the parameterization is refined further after the initial configuration

is finished. First, the DFG is traversed forwards from the components that produce constant output bits, starting in the ones closest to the inputs, and the configuration of subsequent nodes is adjusted accordingly. Similarly, in a second pass, the DFG is traversed backwards from all components which discard part of their inputs, this time starting with the ones closest to the outputs, adapting the configuration of preceding nodes.

3.3 Resource Models

The proposed framework uses a divide-and-conquer strategy to derive the resource and power usage associated with a candidate DFG without the need for time-consuming synthesis and placement of the system. First, the candidate DFG is used to model the area consumption in terms of the overall number of implemented FPGA resource units. In a second step, the power consumption of the system is derived from the resource unit count, using an approach similar to HAPE [133]. Figure 3.3 shows an overview of the involved steps and components of the resource modeling flow. The following sections describe the workings of the proposed models and the involved steps in detail.

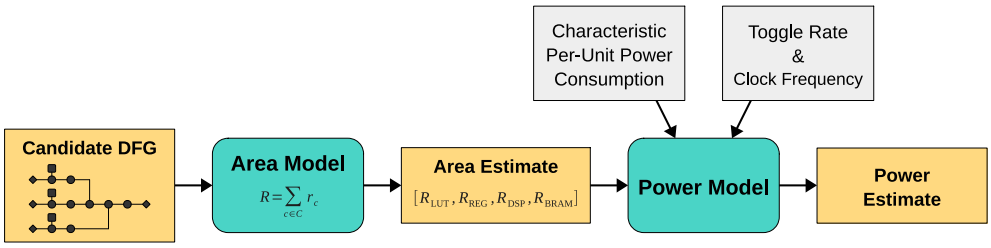


Figure 3.3: Overview of the proposed area and power modeling flow

3.3.1 Area Model

Unlike ASIC implementations, where area consumption is typically assessed in terms of the physical size required by the implemented circuit, e.g. using mm^2 as unit of measurement, the area consumed in FPGA designs is more meaningfully described in terms of the required number of functional units, i.e. LUTs, registers, DSPs and BRAMs. Therefore, the total resource consumption of an application can be denoted by a vector containing the respective counts for all FPGA resource types:

$$R = \left[R_{\text{LUT}}, R_{\text{REG}}, R_{\text{DSP}}, R_{\text{BRAM}}\right]. \tag{3.1}$$

The proposed approach models the required overall FPGA resources R as the sum of the consumption of individual components $c \in C$ in the DFG:

$$R = \sum_{c \in C} r_c, \tag{3.2}$$

where r_c denotes the vector of individual resource units required by the respective component and the sum is calculated element-wise for each resource type. To obtain the values of r_c for all components, the model traverses the candidate DFG and calls the `report_area()` method on each component in the graph, which returns the number of implemented resources depending on the component parameterization.

To test the accuracy of the area model, we randomly created 100 approximate configurations of the channel mixer, using approximations as indicated in Figure 3.1. If the effects of synthesis optimization (see Section 3.2.2) are not taken into account, the estimation of LUTs and registers differs from the actual synthesis results on average by 2.56% and 14.34%, respectively. In contrast, after the parameter refinement introduced in Section 3.2.2, the average relative estimation error was reduced to 1.89% for LUTs and 1.07% for registers.

3.3.2 Power Model

After the resource count of the candidate DFG has been estimated, the results can be used to derive the power consumption of the system. Our approach is based on a method that was proposed as part of the HAPE framework [133]. The main principle behind the model is to acquire characteristic per-unit power consumption values for each resource type and multiply those with the number of required resource units. All FPGA resources contribute a static as well as a dynamic part to the overall power consumption, which are modeled separately.

The overall static power consumption P_{static} of any resource type $\in \{\text{LUT}, \text{REG}, \text{DSP}, \text{BRAM}\}$ depends solely on the number of implemented units R_{type} and is calculated as

$$P_{\text{static, type}} = R_{\text{type}} \cdot Q_{\text{static, type}}, \tag{3.3}$$

where $Q_{\text{static, type}}$ is the characteristic static power consumption per unit of the resource type.

In contrast, the dynamic power consumption scales with the switching activity in terms of signal transitions per second, which can be derived as $\alpha \cdot f_{\text{clk}}$ from the average global toggle rate α and the clock frequency f_{clk}. For any resource type, given its characteristic dynamic power consumption per MHz $Q_{\text{dynamic, type}}$, the dynamic power consumption is derived from the estimated number of units R_{type} as

$$P_{\text{dynamic, type}} = R_{\text{type}} \cdot \alpha \cdot f_{\text{clk}} \cdot Q_{\text{dynamic, type}}. \tag{3.4}$$

Finally, the total power consumption is given by the sum of the static and dynamic power consumption over all resource types:

$$P_{\text{total}} = \sum_{\text{type} \in \{\text{LUT, REG, DSP, BRAM}\}} \left(P_{\text{static, type}} + P_{\text{dynamic, type}} \right) \tag{3.5}$$

While the clock frequency f_{clk} can directly be set by the designer as desired, the other parameters must be determined first as described below.

3.3.2.1 Toggle Rate

The toggle rate depends on the structure of the application as well as the processed input data. Because re-computing exact toggle rates for every candidate solution is time-consuming, we estimate a common global toggle rate by feeding the accurate design of the target application with typical input data. As we target image processing applications, we choose a high-quality image set [134] to represent typical inputs, which can be seen in Appendix A. At first, the input image is converted to a pixel stream in raster-scan order, which is how images are typically fed to stream processing pipelines [14]. Then, the behavioral DFG simulation (see Section 3.1) of the reference application design is used to calculate all internal signals and the bitwidth of each signal is extracted from the DFG. Using this data, a bit vector is generated for each bit position in each internal signal, tracking

the values of that specific signal bit over time. Finally, the global toggle rate is aggregated by calculating the switching activities in all bit vectors and taking their average. For example, considering the DFG of the exemplary channel mixer, the mean internal activity ranges between 0.29 and 0.37 among the individual images in the set, with an average of 0.34.

3.3.2.2 Characteristic Per-Unit Power Consumption

The characteristic per-unit power consumption for the different FPGA resource types depends on the selected target device, its operation temperature and application-specific factors such as the amount of routing required to connect the resources. We propose two different methods to obtain these values, a generic model and an application-specific model:

Generic Model The generic model, which was used in our previous publications [2–6], uses vendor tools to extract application-agnostic per-unit power consumption values which can provide a fair estimate regardless of the application. FPGA vendors provide specific tools, e.g. the Intel Early Power Estimator (EPE) [135] or the Xilinx Power Estimator (XPE) [136], which allow the estimation of overall power consumption based on the number of instantiated units. To include the information contained in such tools into our framework, we extract the respective values for $Q_{\text{static, type}}$ and $Q_{\text{dynamic, type}}$ for all resource types from the respective tool. While the target device and operating temperature can be set directly in the tools, the routing factor is unknown and therefore set to a standard value.

Application-Specific Model While the generic model provides a reasonable estimate that is generally capable of guiding the DSE [4], it does not possess any knowledge about the internal structure of the application and the related routing effort which influences the power consumption. We therefore propose to incorporate application-specific data into the characteristic per-unit power consumption values in order to improve the power estimation with respect to a specific target application. To achieve this, we use the reference configuration together with a few randomly generated approximate configurations for the target application and fully synthesize, fit and route the respective designs. Then, the power analysis tool provided by the FPGA vendor, e.g. the Power Analyzer from Intel [137], is used to generate a power report of the final system which accounts for routing effects as well. From the report, the overall power consumption per resource type is extracted and divided by the respective resource count to derive the per-unit consumption value. Finally, the obtained values are averaged across the sampled system configurations. Using this methodology, an application-specific baseline can be extracted at the cost of only a few synthesis, fit and route runs, which can be reused for all other configurations probed during the DSE.

We compared the accuracy of both models in estimating the power consumption of the exemplary channel mixer using the same 100 random configurations as for the area model analysis in the previous section. Figure 3.4 plots the reference power consumption of each configuration, obtained after synthesis, fit and route in comparison to the values obtained using the proposed power model with the generic as well as the application-specific model for the characteristic per-unit power consumption. For the application-specific model, the reference configuration plus 4 additional random configurations were used to generate and extract the characteristic data. We ordered the configurations according to their reference power values to improve the readability of the data presentation.

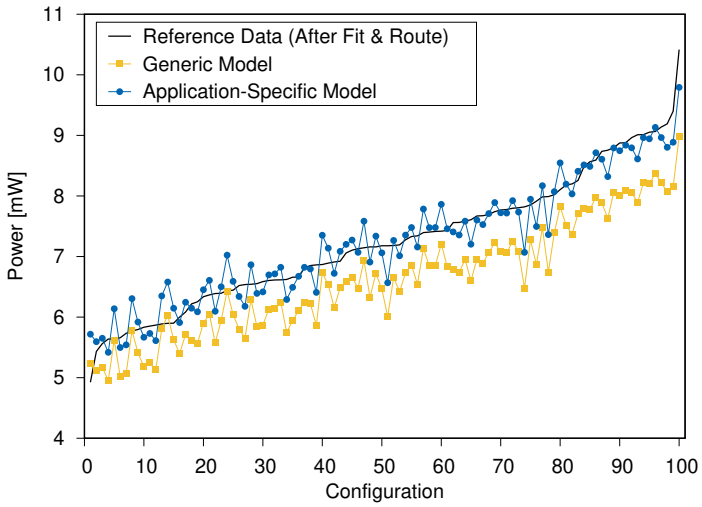


Figure 3.4: Comparison of generic and application-specific power model for 100 random channel mixer configurations

The plot shows that both models follow the overall trend in the power consumption without the need for time-consuming synthesis, fit and route for every configuration. However, the application-specific model follows the reference curve closer and delivers a significantly more accurate estimate of the power consumption. Across the 100 random configurations, the average relative estimation error of the generic model is 8.57% which is reduced by the application-specific model to 3.13%. Although the improved accuracy comes at the cost of characterizing a small number random configurations in the beginning, this overhead might be negligible considering that the number of candidate solutions probed during the DSE is typically higher by several orders of magnitude. Generally, while the generic model is a reasonable indicator for the achievable reduction in power consumption, which could be useful for initial experiments, the application-specific model should be preferred when running the actual DSE for a target application.

3.4 Quality Model

In any approximated system, the degradation in application quality is the drawback that needs to be accepted to achieve benefits in resource usage. When choosing a suitable configuration, the quality estimation needs to enable the designer to judge the expected application quality reliably. This requires the reported values to be suitable for the target application and interpretable by the designer. Consequentially, the quality model used in the proposed framework allows the designer to freely choose a suitable and familiar reference metric.

This section starts with reviewing the state-of-the-art for quality estimation in the field of approximate computing and the drawbacks of commonly used quality models. Then, the functionality of the proposed DFG-based reference metric calculation is explained. Finally, we provide two methods to select appropriate training data for the estimation, and demonstrate their benefits and drawbacks depending on the considered target quality statistic.

3.4.1 Related Work

Within the field of approximate computing research, quality is often measured in terms of basic signal fidelity metrics such as the numerical error distance, the hamming distance or the error rate. Metrics like these are able to capture general potential trade-offs between resource consumption and accuracy. However, they are application-agnostic, which means that they might be of limited relevance for the designers and users of specific real-world applications. Hence, application-specific metrics should be the preferred method for quantifying application quality [17].

For many application domains, research efforts have over decades developed and matured specific metrics which are well understood by designers with domain knowledge. As an example, for signal and image processing applications, quality is often reported by means of the peak signal-to-noise ratio (PSNR). Within the field of image processing, metrics are often even more specialized depending on the nature of the specific application. The quality of image compression and filtering, for example, is often quantified with the structural similarity measure (SSIM) [138], while color accuracy is typically measured in terms of the CIELAB ΔE difference [139]. These and other metrics specifically designed for image processing often take properties of the human visual system into account to improve the prediction of observed quality. Ideally, application designers should have the flexibility to choose their preferred application-specific metric so that they can make well-informed decisions about the quality of an approximated application.

In approximated applications, error propagation across system components needs to be accounted for in the employed quality model. Dealing with this problem, several analytical error propagation models for approximate computing have been proposed in recent years [140–143]. Unfortunately, such methods are generally limited to linear systems and can typically only calculate the error propagation across adders and multipliers. On the other hand, many image processing systems include point operations which apply non-linear transfer functions to pixel values, e.g. using pre-computed look-up tables [118], restricting the usage of current analytical models. Additionally, the output of such methods yields overall signal deviation statistics or error distributions that cannot easily be transformed into many of the established application-specific quality metrics for meaningful interpretation. As an example, in the ΔE metric, the numerical signal differences are weighted differently depending on the output color to consider the varying sensitivity to color changes across different colors in the human visual system [139].

3.4.2 DFG-Based Quality Estimation

The proposed framework uses so-called reference metrics that compare the output of an approximated system to the golden output of the approximation-less reference system. This enables the choice of many application-specific state-of-the-art quality metrics such as the PSNR, the CIELAB ΔE, the SSIM and many others, which fall into the category of reference metrics. To integrate any metric into the flow of the proposed framework, the designer only needs to provide the handle to a function which calculates the respective condensed quality value(s) given the approximate as well as the golden output. By nature, reference quality metrics require a suitable training data set which is used as input from which both the golden and the approximated outputs are computed. Targeting the applications studied in this thesis, we propose two types of training data sets, namely systematically generated *synthetic* inputs and randomly sampled *real-world* inputs. Considerations for the choice of a suitable and relevant training data set will be discussed in the next section.

The proposed quality estimation flow is illustrated in Figure 3.5. First, a suitable training data set is selected based on the application domain and the prominent quality objective. After the training data is chosen, it is processed once with the approximation-less version of the application to obtain the golden output. Then, both the training data and the golden output are stored with the model in preparation of the DSE.

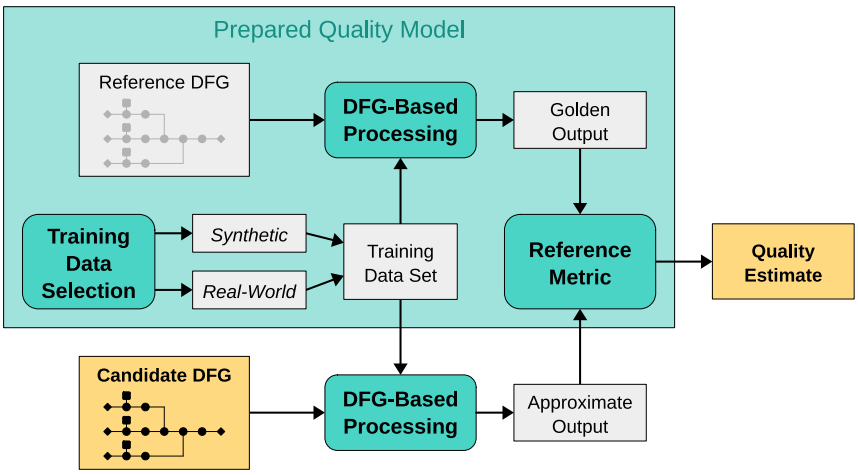


Figure 3.5: Overview of the DFG-based quality estimation process

For each potential solution probed during the DSE, the corresponding candidate DFG is used as described in Section 3.1 to process the training input data, returning the respective approximate output. Since the employed DFG-based processing is a bit-accurate end-to-end simulation of the parameterized hardware system, it implicitly accounts for error propagation effects. Finally, the provided reference metric function is called to obtain the desired quality estimation results.

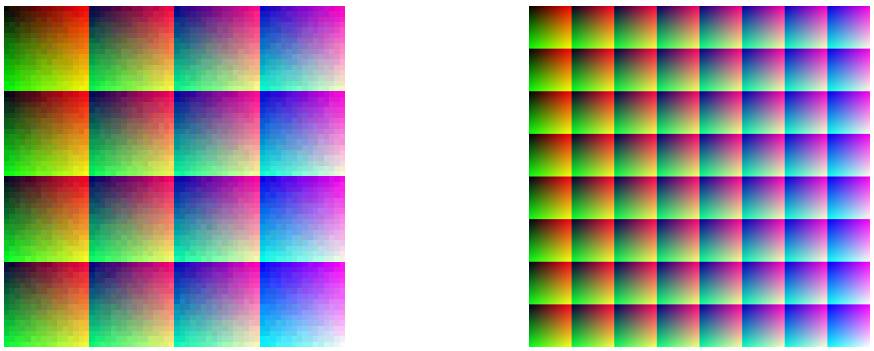
3.4.3 Choice of Suitable Training Data

Like the choice of the quality metric itself, the selection of suitable training data should be related to the target application, so that relevant inputs are covered. In this section, we will concentrate on the selection of relevant training data for the case studies presented in this work, which process image color data. However, the general principles used in the selection process may be applied to different application types as well.

When optimizing application quality, there are generally two slightly different potential objectives: (a) guaranteeing a defined worst-case quality degradation bound and (b) ensuring a specific average output quality level. Existing reference quality metrics reflect these targets to different extends. For example, while penalizing large errors due to its internal use of the MSE, the PSNR largely captures the average quality loss for a given data set. On the other hand, both the mean and/or the maximum ΔE aggregated across the data set can be used to determine the average and/or worst-case color shift in the output. The target objective type, however, also influences the desired properties of the training data set necessary for a meaningful and accurate quality estimation.

In the following, we motivate and propose two methods for selecting suitable training data sets for image color processing applications, each specifically targeting one of the potential objectives described above.

Synthetic Training Data Sets In the first case, for limiting the worst-case degradation in quality, the training data should cover the entire input space, including potential corner cases. For pixel color data, ideally all possible input colors are included in the training set. The overall number of possible colors scales exponentially with the image bitdepth, i.e. the number of bits used to represent a single color channel. In professional cameras, 10 or more bits are commonly used per channel which leads to very large input ranges [144]. The applications studied in this work use a common depth of 12 bits per channel, corresponding to $2^{36} = 6.872 \times 10^{10}$ possible input colors. Hence, the respective exhaustive training data set would need more than 288 GiB of data and the calculations for the quality estimation would be extremely time-consuming. To overcome this problem, we propose to create a *synthetic* image color training set by sampling the input space in regular intervals along each dimension. This ensures that colors across the entire input space are covered while strongly reducing the number of actual inputs in the training set. However, perfectly equidistant sampling would lead to regular patterns in the training set, e.g. in some cases the LSBs of all pixel values would be set to zero. To prevent the optimization from adapting to such patterns, we add uniformly distributed noise between the sampling steps so that the colors are slightly shifted to irregular positions. The training set size can be adapted by choosing the number of sampling steps to control the trade-off between effort and accuracy of the estimation. Figure 3.6 shows the resulting training sets for two different sampling distances, using 16 respectively 64 sampling steps along each dimension and therefore containing $16^3 = 4096$ and $64^3 = 262\,144$ inputs.



(a) 16-step sampling (b) 64-step sampling

Figure 3.6: Synthetically generated training data with regular sampling of color space

Real-World Training Data Sets Secondly, when the property of interest is the expected average output quality of the approximated application, then the training set should contain the inputs that are most commonly seen. The regular sampling introduced above contains a balanced mixture of colors and does not necessarily represent the typical distribution of colors when shooting real-world scenes. Another factor besides the actual image content that has a significant impact on the typical distribution of image colors is the color space and luminance encoding or *image state* [145]. Two image states commonly used in digital imaging workflows are the *scene-referred encoding* and the *output-* or *display-referred encoding*. The scene-referred encoding aims at relating the encoded values as closely as possible to the color characteristics of the objects in the captured scene. In practice, luminance is typically encoded logarithmically to retain high dynamic range (HDR) information, which is inspired by the behavior of negative film used in analog pho-

tography, and the colors are transformed to a wide-gamut color space [134]. On the other hand, the display-referred encoding is linked to a specific display device or print medium and aims at providing an intended appearance given the characteristics of the target device. These characteristics are often standardized for a specific class of devices, e.g. in the *Rec709* standard for high definition television (HDTV) devices [146]. Figure 3.7 illustrates the difference between scene-referred and display-referred encoding for the *Color Wheel* image from the ARRI image set used throughout this work [134]. Because the scene-referred encoding uses a larger container to retain even extreme real-world colors and luminances, most pixel values in an evenly exposed scene are within the middle of the range, which is why the image appears as having low contrast. On the other hand, in the display-referred encoding which typically represents a limited dynamic and color range, the values of a typical image cover a larger portion of the numerical range.

(a) Scene-referred encoding (logC) (b) Display-referred encoding (Rec709)

Figure 3.7: Image *Color Wheel* from the ARRI Image Set in scene-referred (LogC) and display-referred (Rec709) encoding [134]

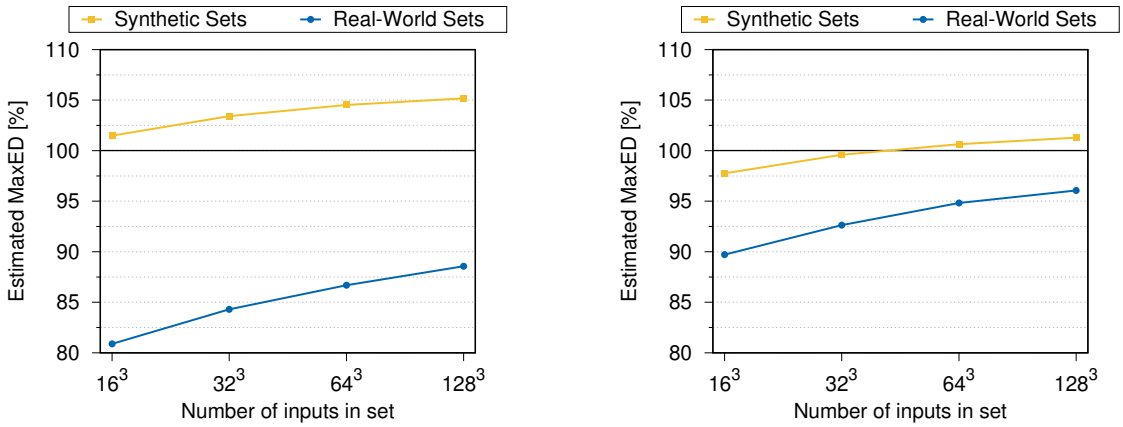
This is why the color encoding should be taken into account when selecting training data for estimating the expected average quality loss. To achieve this, our methodology creates real-world training data sets by randomly picking input colors from one or more user-supplied sample images which are encoded in the correct image state as needed for the application. By choosing the sample images, the designer can control which types of colors are covered in the training set and using the correct encoding ensures that the estimation focuses on the relevant ranges of the input space. Furthermore, the size of the training set can be controlled directly by changing the number of random draws when collecting the data.

Comparison of Training Data Sets

To analyze the impact of training data set type and size on the accuracy of the quality prediction, we generated synthetic as well as real-world training sets of different sizes as described above. For the synthetic sets, we used 16, 32, 64 and 128 sampling steps along each dimension, leading to sizes from 16^3 to 128^3 inputs, respectively. The real-world sets use randomly sampled pixels from the *Color Wheel* image of the ARRI image set and the generated sets have the same sizes as the synthetic sets for comparison. We use the 11 other images from the image set (see Appendix A) to generate ground truth data from real-world use cases to validate the quality estimation. Note that the resolution of the images is 2880×1620 and therefore the validation set contains roughly 51.32×10^6 inputs which is more than 24 times larger than the biggest training set and more than 12 500 times larger than then smallest training set.

The validation is carried out over the same 100 random approximate configurations of the channel mixer that were used above in the analysis of the resource models. We use the MaxED to represent the worst-case quality loss and the PSNR to represent the expected average quality. To show the influence of the color encoding, we repeated the analysis for both the scene-referred encoding (logC) and the display-referred encoding (Rec709) of the validation set. The generation of the real-world sets respectively uses the same encoding when sampling inputs from the *Color Wheel* image.

Estimation of the worst-case error First, we analyze the performance of the training sets for estimating the worst-case quality bound. Hard guarantees for the actual worst case error can only be given if all possible inputs are tested, which is infeasible for higher bit depths as described above. When using a reduced training set, testing more inputs leads to a higher probability of finding the critical ones that lead to high errors. This leads to a trade-off between the certainty of the estimated error bounds and the related computational effort. Figure 3.8 plots the estimated MaxED for all combinations of training set type and size for both studied encodings. The values are reported relative to the respective worst-case error found across all images of the validation set and averaged over all 100 random approximation configurations.



(a) Scene-referred encoding (logC)

(b) Display-referred encoding (Rec709)

Figure 3.8: Average estimated worst-case error relative to ground truth for scene-referred and display-referred encoding of the validation set

In the plot, values above 100% mean that, on average, the worst-case error found in the validation set is within the bounds estimated using the respective training set. On the other hand, values below 100% indicate that the quality model tends to underestimate the maximum error. As expected, the plot shows that increasing the number of inputs in the training set also increases the estimated worst-case error in all cases. However, the worst-case errors found in entire real images significantly exceed the predictions from using the real-world training sets in all cases. The synthetic sets, in contrast, yield more reliable bounds. Nevertheless, their predictions seem to be more robust when the inputs are given in scene-referred encoding compared to the display-referred encoding. In contrast, the predictions of the real-world sets are more accurate for inputs in display-referred encoding, which leads to a smaller difference between both data set types in that case (see Figure 3.8b in comparison to Figure 3.8a).

Estimation of average quality Secondly, we compare the capability of the different training sets in estimating the expected average quality in terms of the PSNR. Table 3.1 reports the model accuracy in terms of the relative error in predicting the mean PSNR all images in the validation set, averaged across all 100 random configurations of the channel mixer. Given are values for all generated training sets and for both studied color encodings. In contrast to the estimation of worst-case error bounds, the size of the training data has negligible influence on the prediction accuracy in this case. However, the real-world data sets show a better accuracy in predicting the average quality, especially for the scene-referred encoding, where the prediction error is reduced by more than half compared to the synthetic sets.

Table 3.1: Accuracy of predicting PSNR using different training sets for scene-referred and display-referred encoding of the validation set

Encoding	Average relative prediction error [%] when estimating PSNR using							
	Synthetic Sets				Real-World Sets			
	16^3	32^3	64^3	128^3	16^3	32^3	64^3	128^3
Scene-referred (logC)	8.99	9.00	9.00	9.00	4.43	4.43	4.45	4.44
Display-referred (Rec709)	9.10	9.10	9.10	9.10	6.79	6.75	6.75	6.74

With the proposed methods, we provide the designer with different options to select training data for color processing applications such as the ones studied in this work. The results of the analysis for the exemplary channel mixer show that data sets covering the entire input space are suited better to extract error boundaries while focusing the selection towards the most probably inputs enables a better estimate of the expected average quality. Similar deliberations should be made in the selection of training sets when targeting other types of applications.

Depending on the most prominent target objective and the available budget of computational resources, the designer can choose the fitting combination of selection method and training set size. Furthermore, it is possible to combine both methods if both objectives are similarly important. In that case, the approximate output can be generated for both training sets and each output is used to calculate the reference metric linked to the respective quality objective.

3.5 Chapter Summary

This chapter presented the proposed methodology for managing the integration and parameterization of different approximation methods within target applications and for modeling the quality-resource trade-off on the application level. In the proposed framework, any target application is structurally represented by its DFG, and graph annotations are used to integrate the approximations and define the design space. This representation also facilitates dealing with parameter dependencies between components. Furthermore, resource models were presented which estimate the area consumption with a divide-and-conquer approach using the fully parameterized candidate DFG while the power usage is derived directly from the required number of FPGA resource units. It was shown that the proposed approach is able to accurately predict resource consumption without the need for the time-consuming synthesis, fitting and routing procedures. Lastly, a quality model was proposed which allows the designer to choose and easily integrate their preferred reference metric

to obtain relevant and interpretable application-specific quality estimates. Implications regarding the selection of suitable training input data were discussed and two specific data set types, a synthetic set and a real-world set, both tailored specifically towards the target application scope, were proposed and evaluated with regard to different quality statistics. The evaluation has shown that the synthetic set should be preferred for estimating worst-case error bounds while the real-world set performs better when estimating average quality loss.

Chapter 4

Design Space Exploration

Combining multiple parameterizable approximations within a target application leads to very large design spaces which grow exponentially with the number of exposed parameters. The parameter dependencies between system components and the effects of error propagation necessitate a joint global optimization of all parameters. Furthermore, the relation between the parameterization and both the resource consumption and the application quality is highly non-linear and cannot directly be described by analytical functions. Lastly, since approximated applications trade off the competing goals of resource savings and application quality, the DSE is a multi-objective optimization problem by nature. Consequentially, instead of a single optimal solution, the search yields a multitude of Pareto-optimal solutions from which the designer can choose the parameterization associated with the most suitable trade-off according to their preferences. Altogether these properties of the optimization problem call for an appropriate search heuristic to efficiently explore the design space.

Such a search heuristic typically operates in a generate-and-test loop as depicted in Figure 4.1. At the beginning of each cycle, one or more candidate solutions are generated which are then evaluated in terms of their fitness according to the target design objectives. In approximate system design, the fitness of a candidate solution quantifies its quality-resource trade-off. Finally, the loop is closed as the fitness estimates are fed back into the search heuristic which tries to utilize the information to generate improved candidates. Ultimately, the DSE finishes once a stop condition is reached, which could be a predefined number of iterations or a convergence criterion. The final output of the DSE is given as the list of candidates whose fitness is Pareto-optimal in the multi-dimensional objective space among all probed candidates.

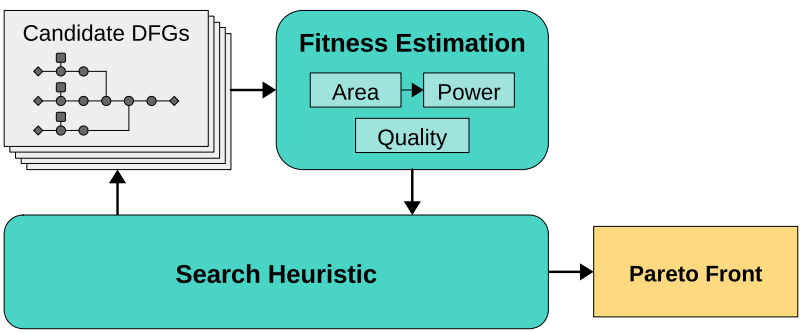


Figure 4.1: Overview of the DSE process

This chapter starts with an overview of proposed approaches for exploring the design spaces of approximated systems found in related literature. Then, the use of the GA within the proposed framework is motivated and its core functionality described. Lastly, we describe how the GA is adapted for and used within the proposed framework to deal with the peculiarities of guiding through the DSE for approximated systems. The optimization methodology employed in the framework as described in this chapter is based on work by Manuel et al. [5].

4.1 Related Work

Within the research field of approximate computing, several approaches for exploring the design spaces of approximate hardware-based systems have been proposed in recent years, which are discussed in this section. These approaches differ in the employed exploration methods as well as in the types of approximations they support.

Starting from a behavioral description of the target circuit, *ABACUS* creates an abstract syntax tree (AST) representation of the design and applies various approximate transformations, such as scaling of intermediate signals, replacement of arithmetic operations or variable-to-constant substitutions [147]. For the DSE, they employ a stochastic greedy algorithm, which starts with the original design and evolves it through a defined number of iterations. In each iteration, the algorithms tests a number of randomly picked approximate AST transformations, employing gate-level simulations and synthesis for the fitness estimation. Then, the best candidate is chosen according to a weighted linear combination of the estimated accuracy as well as the area and the power consumption. The best candidate then serves as parent for the next iteration. However, their approach does not discuss how suitable weights for the linear combination can be chosen. Furthermore, picking only a singular best design as the seed for the next iteration limits the search and may lead to a sub-optimal exploration. To overcome some of the drawbacks in the original design, the approach was later extended to include a hybrid selection scheme, in which the linear-scaled fitness ranking is combined with the multi-objective Nondominated Sorting Genetic Algorithm-II (NSGA-II) scheme [148] to select multiple parent designs for the next generation [149]. Additionally, the fitness estimation itself is sped up by including an accelerated, C-based simulation workflow and by parallelizing the fitness estimation of all candidates. However, the hardness of defining suitable weights for the linear fitness ranking as well as the need for time-consuming per-candidate synthesis remains.

Specifically targeting high-level synthesis (HLS) workflows, Xu and Schafer proposed an approach which applies approximations in different phases of the design flow [150]. At the software-level, infrequently executed code lines are pruned and variable-to-variable and variable-to-constant substitutions are applied. In the next phase, the arithmetic units are approximated in two steps. First, the impact of any possible individual unit replacement is characterized. Secondly, a greedy algorithm iteratively adds approximate unit replacements to the design, minimizing a linearly weighted combination of the competing objectives accuracy and resource consumption, constructing a trade-off curve of Pareto-dominant solutions that have different numbers of approximated operations. The last phase finally repeats the variable-to-variable and variable-to-constant substitutions at the register-transfer level (RTL), where all internal signals are available, and additionally uses a profiling step to substitute individual bits with constants. While the first two phases rely on model-based HLS synthesis, the third phase includes the need for time-consuming logic synthesis. Also, the sequential application of the different approximation methods in subsequent phases

might leave some important regions of the design space unexplored. Furthermore, their methodology does not include a model of power consumption.

Zervakis et al. propose a technique to integrate multi-level approximate arithmetic units, i.e. units that employ combinations of functional approximation and VOS, into a target application [24]. Their approach generates a number of random training configurations of the target application which are simulated to obtain the associated output error values. Then, a neural network is trained to facilitate the quality estimation for arbitrary configurations and the power consumption is estimated by summing the characteristic individual consumption of all components. With these models, the quality-resource trade-off is exhaustively evaluated for all possible configurations and the Pareto front is extracted from the results. As a final step, a heuristic voltage island formation algorithm is proposed to limit the number of different supply voltages. This approach targets ASIC designs and includes locally applied VOS, which limits its applicability for FPGA designs. If VOS is removed to make their approach work on current commercial FPGAs, the approach becomes limited to approximate arithmetic units only.

Mrazek et al. proposed a methodology called *autoAx* to find Pareto-optimal combinations of approximate circuits from a predefined library to replace accurate operations in a target application [151]. First, the application is profiled to obtain the input distribution at each operation. Using this information, the library is filtered and Pareto-optimal circuits are selected for each operation w.r.t. the hardware cost and the expected error according to the input distribution. Then, similar to the approach described above, random configurations of the application are created, and application-level quality as well as hardware cost values are obtained using simulation and synthesis of these configurations. The resulting data set is used to train ML models so that simulation and synthesis can be omitted during the DSE. A heuristic hill climbing variant is used to construct the Pareto set. The algorithm starts with a random configuration and visits randomly selected neighbors, i.e. configurations that differ in one operation. If the neighbor dominates the current Pareto set, it is added to the set and chosen as the new parent for the next iteration. While the original approach targets ASIC designs, another proposed variant named *ApproxFPGAs* extends the methodology for FPGA-based applications [124]. A drawback of the search heuristic employed in both of these approaches is that it traverses the search space along a singular trajectory and therefore employs a local search which might limit the diversity of the evolved solutions.

AxHLS is another approach that tries to minimize a linear combination of area, power and latency metrics under a given error constraint [152]. To explore the search space, it uses Tabu Search to iteratively select approximate arithmetic units to replace the accurate operations. Similar to the autoAX/ApproxFPGA approach, this algorithm uses a local search and employs models to accelerate the fitness estimation. Another common drawback is the limitation to a singular type of approximations, namely approximate arithmetics.

The methodologies proposed in related work exhibit different deficiencies and none of them completely meets all of the requirements needed for an efficient exploration of large design spaces spanned by the combination of various approximation methods in FPGA designs. ABACUS [147, 149] and the work by Xu and Schafer [150] employ logic synthesis within their workflow which hinders the use of their methods for very large design spaces. On the other hand, the work of Zervakis et al. [24] and both autoAX/ApproxFPGA [124, 151] and AxHLS [152] support large design spaces by employing fast models for fitness estimation. However, all of them restrict the design space to approximate arithmetic units (for the multi-level approach by Zervakis et al. [24] this holds true when targeting FPGA implementations, which excludes the application of local VOS).

4.2 GA-based DSE Approach

Current state-of-the-art DSE techniques for configuring approximate hardware predominantly apply heuristics that search locally and follow a singular trajectory through the design space by successively modifying one parameter or component in each step. Furthermore, many of the proposed methods condense the quality-resource trade-off into a singular fitness value using a weighted linear combination of the competing objectives in order to identify the best direction for moving forward. Although the weights for the objectives allow the DSE to incorporate designer preference into the search, their balanced scaling may not be straightforward. These properties might limit the diversification of the results and narrow the explored region within the design space, potentially restricting the exploitation of benefits offered by combined approximation methods.

To overcome such limitations, the proposed framework employs a metaheuristic approach using the genetic algorithm (GA) that does not require prior knowledge about final optimality or the impact of individual parameters. Furthermore, it directly incorporates multi-objective optimization into the search. During the search, a population of candidate solutions is evolved which allows the algorithm to balance between selection pressure and candidate diversity, inherently combining trade-off exploitation and global exploration in the search.

The rest of this chapter presents the employed DSE methodology, which is based on [5]. First, an overview of the functionality of the GA is given. Then, the used GA setup and necessary adaptions of the individual GA components for a purposeful integration into the proposed framework are described in the subsequent sections.

4.2.1 Overview of the GA Process

The GA can be classified as an evolutionary algorithm variant [153]. It operates in a loop to gradually evolve and improve a population of candidate solutions, in contrast to local search algorithms which iteratively improve a single candidate. In the GA, the candidates are represented using a genetic encoding that maps the real-world properties of the original problem context to a form that can be purposefully manipulated by genetic operations to evolve them. These encoded candidate solutions are commonly referred to as *chromosomes* or *individuals*. Figure 4.2 shows a high-level overview of the optimization flow and the involved steps.

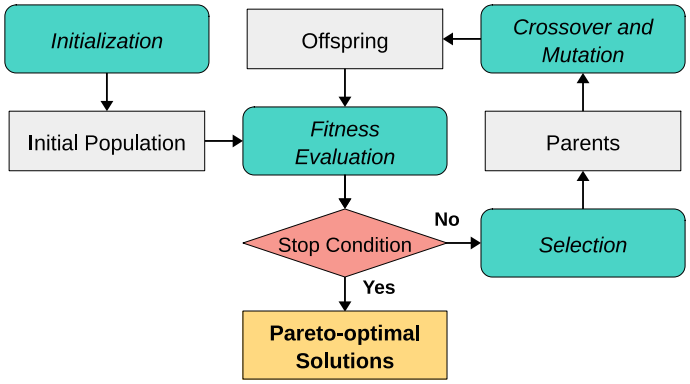


Figure 4.2: Generational loop of the GA-based optimization flow

The process starts with the *initialization* phase in which a first population of individuals is generated randomly. However, the generation function must be designed to ensure feasibility of the created individuals, i.e. adhering to constraints imposed by the effects of parameter dependencies, as discussed in Section 3.2. The population then enters the optimization loop where, as a first step, the fitness of each individual is evaluated in the objective space. For this, all individuals are decoded to form the respective parameterized candidate DFGs, which are used by the resource and quality models proposed in Chapter 3 to estimate their fitness. Instead of aggregating all objectives into a single value, they are kept individually. Next, a *selection* among the individuals of the population takes place to choose μ parents as mating pool from which the offspring will be evolved. Informed by the fitness values, the selection ensures a survival of the fittest individuals and is responsible for pushing improvements in the evolution. In the following step, genetic operations, i.e. *crossover* and *mutation*, are applied to the parents to produce a new generation of λ offspring individuals. The crossover operation describes a binary variation that merges genes from two parents to create an offspring individual. On the other hand, mutation is a unary operation in which the offspring is generated by modifying a small part of a single parent. The genetic operations operate stochastically, creating random, unbiased modifications (mutation) respectively genetic mixtures (crossover). Finally, the fitness evaluation of the newly evolved offspring individuals then starts the next iteration of the loop. This loop iterates until a stop condition is satisfied, which can be based on convergence analysis, reaching a desired sufficient fitness level, or exhausting a pre-scheduled computational limit, which is often defined by the number of evolved generations. In the proposed framework, we set a fixed number N_{gen} of iterated generations as stop condition after preliminary analysis of the convergence for each case study. Once the stop condition satisfies, the DSE returns the evolved set of configurations that are Pareto-optimal among all visited candidates across all generations. In the proposed framework, the core functionality of the GA-based DSE is implemented using the DEAP framework[1] [154].

4.2.2 Encoding and Genetic Operations

The genetic operations that work on the encoded individuals need to consider the parameter dependencies to ensure the generation of feasible solutions. To facilitate the definition of these operations, the employed encoding scheme describes the individuals using nested lists of real-value genes. The lists' hierarchy defines the dependency of the parameters. Each sub-list describes a section of the DFG with inter-component parameter dependencies. Hence, by definition, there are no parameter dependencies between the parameters of two separate lists at the same hierarchy level. Furthermore, the individual lists and the parameters within them are ordered according to their parameterization order as discussed in Section 3.2.1.

As a generic example, the list

$$\Big[\big[A_1, A_2, A_3, A_4\big], \big[B_1, B_2, B_3\big]\Big] \tag{4.1}$$

contains two parameter groups, A and B. The parameters A are independent of the parameters B, and their indices capture the order of parameterization.

This structure of the encoding enables a straightforward way to locate safe split points for the genetic operations. Given this encoding, a generic one-point, n-point or uniform crossover operation

[1] `https://deap.readthedocs.io/`

[153] can be executed after restricting the choice of split points to lie between independent chromosome sub-lists. That is, the generated offspring uses a new combination of sub-lists sampled from both parents. Nevertheless, the designer can provide custom functions to enable splits inside individual lists, if desired. A practical example for this is given for the crossover of two sparse table parameterizations as used in two of the case studies presented in the next chapter, which is described in Section 5.3.3.

Regarding the mutation operation, there are two generic options for safe execution. Since the individual sub-lists are independent of each other, the mutation can randomly pick any sub-list and completely replace it with random regeneration, adhering to the defined parameterization order, of course. Secondly, to mutate only part of a sub-list, the operation can randomly choose a split point within the list and only regenerate the parameters after the split point.

4.2.3 Selection

The driving force that moves the evolutionary optimization towards better solutions is the selection process. During this process, a mating pool of μ parents needs to be selected, which seeds the creation of the next offspring generation. The proposed framework employs a modified variant of the Nondominated Sorting Genetic Algorithm-II (NSGA-II), which is an established selection method that directly handles multiple objectives and balances selection pressure and diversity among the individuals [148]. Figure 4.3 illustrates the standard NSGA-II selection procedure. To introduce elitism into the selection, a $(\mu + \lambda)$ strategy is used, i.e. the selection of new parents is taken among a combined population of the current λ offspring individuals and the last parent population of size μ from which they evolved.

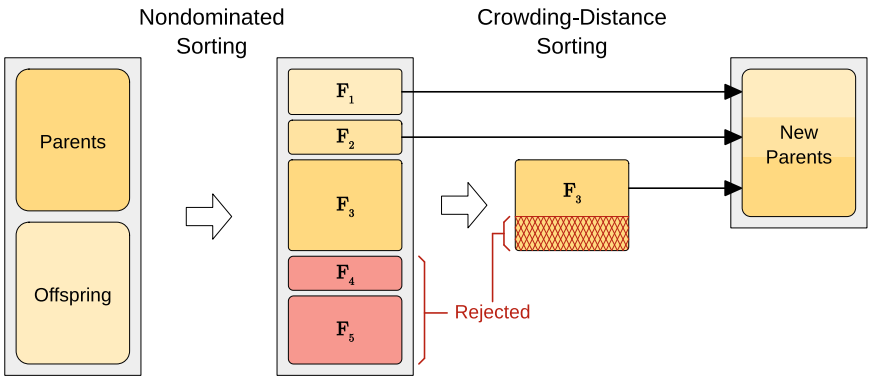


Figure 4.3: NSGA-II selection procedure (adapted from [148])

The selection procedure has two major steps. First, all considered individuals are sorted into multiple nondomination fronts F_r according to their dominance rank r, i.e. individuals in front F_j are only dominated by those with a lower dominance rank $i < j$, and the individuals in the first front F_1 are Pareto-optimal among all considered individuals. In the second step, μ new parents are selected from the fronts in ascending rank order, starting from F_1. As long as the number of individuals in the next front F_i is less than the number of remaining individuals to select, the entire front is added to the selection. However, when the next front contains more individuals than the remaining number required to complete the selection, the members of that front are ranked inter-

nally to enable a meaningful choice among them. We will further denote this front as *final front*. Here, the classical NSGA-II employs a crowding-distance sorting which calculates the distances between adjacent individuals in the objective space and ranks those in less crowded regions higher than those surrounded by other individuals of similar fitness. While the initial selection according to nondomination rank guides the optimization towards overall better solutions and therefore convergence, the crowding-distance sorting ensures diversity among the population.

ROI-Based Nondominated Sorting Selection (ROI-NSGA)

The selection method of the NSGA-II leads to an efficient exploration across the entire search space. However, this is not always necessary in practical applications. Even though different levels of application quality might be acceptable if they offer different favorable quality-resource trade-offs, there is often a minimum quality threshold below which the trade-off becomes irrelevant to the designer. Similarly, depending on the available area or power budget, the designer may not be interested in solutions that consume resources above a certain maximum threshold. Such boundaries define a region of interest (ROI) in the objective space into which the search should be focused.

To achieve such a concentration into a certain objective region, the framework employs a modification of the NSGA-II selection methodology named Region of Interest Nondominated Sorting Genetic Algorithm (ROI-NSGA) [5]. As input, it takes a user-defined ROI in the objective space. The selection procedure is the same as with NSGA-II, except for the choice within the final front that contains more individuals than still needed. While the NSGA-II uses the crowding-distance measure to rank these individuals, the ROI-NSGA differentiates three phases based on the number of individuals from the final front that lie inside the ROI. Depending on the phase, the algorithm switches between different sorting mechanisms as follows:

- **Phase I:** *The ROI contains **less than two** individuals from the final front.* In this initial phase, the selection is based on the crowding-distance as in the standard NSGA-II. This phase continues until the GA identifies at least two solutions from the final front within the ROI.

- **Phase II:** *The ROI contains **at least two, but not enough** individuals from the final front, i.e. the number of solutions within the ROI is less than the number still needed for selection.* A dynamic reference point is calculated, which lies in the middle between the ROI boundary and the best achieved value among all considered points inside the ROI, calculated separately for each objective dimension. Instead of the crowding-distance, this phase uses the distance of each point to the reference point, calculated by the L^1-norm after normalization by the boundary values in each dimension. The algorithm selects the individuals with the minimum distance to the reference point, which serves to invite the solutions in or closest to the ROI to the selection.

- **Phase III:** *The ROI contains **enough** individuals from the final front for the remaining selection.* Since there are enough solutions in the ROI to choose from, the points outside the ROI are discarded. For the selection among the remaining solutions, the algorithm switches back to using the crowding-distance as indicator in order to spread the selection across the ROI and maintain diversity.

With these phases and the different selection methods for the final front, the ROI-NSGA first tries to concentrate the search towards the ROI and then strives to optimally explore the possibilities

inside the ROI. Please note that while the numbering of the phases indicates their typical sequence, depending on the current state of the search, they may alternate in the course of the generational loop. Also, if there are enough points inside the ROI in the random initial population, phase 1 or even phase 2 may be skipped in the beginning.

4.3 Chapter Summary

This chapter discussed the optimization strategy used within the DSE phase of the framework. First, the approaches found in related work are discussed and their deficiencies analyzed. Based on the requirements of the framework, a GA-based metaheuristic optimization strategy was selected to directly handle the inherently multi-objective quality-resource trade-off and to efficiently explore the highly complex design spaces resulting from combining multiple approximations within single applications. In the rest of the chapter, the optimization workflow of the selected methodology was described in detail, including the design of genetic operations and the ROI-based selection process.

Chapter 5

Case Studies

This chapter presents three case studies of image processing applications which are suitable targets for including various approximations to exploit the quality-resource trade-off. These case studies serve to demonstrate and evaluate the utility of the methodology proposed in this thesis for practical applications. They represent real-world processing pipelines that are used to process image colors in digital camera systems. Using the framework presented in this thesis, approximations of different types are integrated into the applications and their parameters are optimized.

These case studies represent varying levels of complexity and focus on different aspects. In the first case study, which contains a single image processing stage used to convert colors from the RGB format into the YC_bC_r representation, the focus lies on the combination of precision scaling and approximate arithmetic units. Next, the second case study uses a series of steps to adapt image colors for display on a monitor with specific characteristics. It builds upon the first one and demonstrates how modules from a simpler application can be reused in a larger one by adding non-linear transformation stages and table-based approximations to the system studied in the first case study. These first two case studies have been published in [4] but were extended for this dissertation to use the full potential of the proposed framework in its current state. Most notably, the related changes cover the integration of approximate multipliers into the matrix multiplication step and adjustments in the pipelining structure to stabilize timing. Finally, the third case study targets a popular approach adopted in many cameras to manipulate image colors, which is color transformation by means of a 3D-LUT.

In all case studies, the use of the full design space, allowing for a combined use of different types of approximation methods that are simultaneously parameterized, is compared to the use of restricted design spaces enabling only a singular type of approximation to evaluate potential benefits of combining different approximation types within the target applications. These restricted design spaces can also be seen as a simulation of the potential offered by related approaches whose methodologies are restricted to specific approximation types.

The rest of this chapter is structured as follows: First, common parameters used in the experimental setup are given in Section 5.1. Then, the three case studies are presented and evaluated one after the other. The purpose, functionality and structure of each application is described first. Then, suitable approximations are selected from the approximate component library and integrated into the respective application, and the resulting design spaces are described and their complexity assessed. After defining the genetic encoding of the parameter space and detailing the related genetic operations, the application-specific experimental setup is given. The results of the DSE are presented and discussed, and the fitness estimation by the proposed models is validated against post-synthesis data as well as the quality seen with real images. Section 5.5 finally summarizes the experimental findings across all case studies and concludes this chapter.

5.1 Common Experimental Setup

The following paragraphs summarize a few common settings used across all experiments related to the presented case studies, including application-related settings, the setup of GA parameters and the general experimental setup configuration.

Application-Related Settings As mentioned in the introduction, the target FPGA device is the `10AS066N3F40E2SG` model from the Intel Arria 10 device family [15] for all case studies, targeting an operating frequency of 266.66 MHz (which supports the pipelined processing of 4K images at up to 30 frames per second) and an operating temperature of 50°C. In the selection of approximate multipliers, we set the speed safety margin to 10% for the first two case studies and to 15% for the last one, which is more complex. Configurations for which the modeled speed is less than the safety margin above the target frequency are filtered out. The overall color bit depth is commonly set to 12 bits per channel. For all case studies, we set the minimization of the estimated power consumption as a singular objective regarding resource usage for simplicity, as it is directly related to the number of implemented FPGA resources. However, the DSE is able to keep track of the employed LUTs, registers, DSPs and BRAMs to ensure that they do not exceed the capacity of the target device. In terms of estimating application quality, each case study uses a different setup which is explained in the respective sections below.

GA Parameters There are a few general parameters in the setup of the GA, which are set to the same values for all case studies. The probability of applying a crossover operation is set to 0.7 and the mutation probability is set to 0.3. In each generation of the GA loop, $\mu = 50$ parents are selected from which $\lambda = 100$ offspring candidates are evolved. While it is likely that better GA parameters do exist, finding the optimal ones typically requires time-consuming hyperparameter optimization for every individual target application. However, our experiments demonstrate that even with non-optimized standard parameters, the proposed optimization approach is able to evolve meaningful, well-populated quality-resource trade-off fronts across all studied design spaces. Hence, these experiments serve as proof-of-concept for the usability of the chosen GA parameters while their optimal setup might be investigated further in future work.

General Experimental Setup The experiments are run on a workstation computer (Intel Xeon W-2145, 8 Cores, 128 GB RAM). The runtime of the DSE differs between the different target applications and the explored design spaces. Even though we use fast and simple fitness models, the time consumption is dominated by the fitness estimation as the overhead introduced by the other steps of the GA loop is comparatively low. Because the fitness evaluation of individual candidates can be done independently, multiple threads are used to parallelize the evaluation of solutions across any generation. When running the DSE for N_{gen} generations, $\mu + N_{\text{gen}} \cdot \lambda$ candidates need to be probed. Hence, the overall DSE runtime is approximately proportional to the number of evaluated candidate solutions divided by the number of CPU cores N_{cores}:

$$t \sim \frac{\mu + N_{\text{gen}} \cdot \lambda}{N_{\text{cores}}}. \tag{5.1}$$

Due to the non-deterministic nature of the GA-based optimization, which generates different results in each run, we repeated the DSE 30 times for each setup to ensure consistency of the results.

Comparison to related work The applications featured in the presented case studies combine approximations of different types, employing models for a fast estimation of resource consumption. Due to the large design space sizes and the time consumption related to synthesis, fitting and routing for FPGA device targets, a direct comparison to related approaches that employ synthesis as part of their fitness estimation is not feasible [24, 147, 149]. On the other hand, related methodologies proposed for model-based exploration of large approximate hardware design spaces are commonly limited to a single type of approximation, namely approximate arithmetic units [124, 151, 152]. To enable a qualitative comparison of our results to these methods, we run the case studies not only with the full design space but also after limiting to single approximation types. Based on the experimental results obtained from these different design spaces, we discuss the potential implications of such limitations in related work.

5.2 Case Study 1: RGB to YCbCr Conversion

The first case study examined in this work, further denoted as CS1, covers a target application which contains the conversion of image colors from RGB triplets into the YC_bC_r representation, which separates the luminance information, stored in the Y coordinate, from the color information, stored in the color difference values C_b and C_r. The application pipeline is depicted in Figure 5.1.

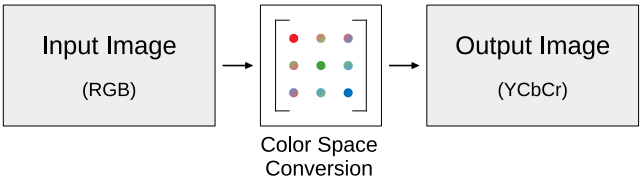


Figure 5.1: Case Study 1: RGB to YCbCr conversion

As shown in the figure, this application consists of one processing stage, in which the input is multiplied with a 3×3 conversion matrix. Consequentially, the application consists purely of arithmetic units. Hence, the focus of this case study is on the combination of precision scaling and approximate arithmetic units. To show the benefits of such a combination across different approximation types, we will run the DSE for different design spaces: (a) the full design space, (b) only using precision scaling, which will be denoted as *precision scaling design space* for simplicity, and (c) only using approximate arithmetics, denoted as *arithmetic units design space*.

This application implements three channel mixer blocks in parallel, which have the same internal structure as the exemplary channel mixer DFG introduced in Section 3.1.

For our experiments, we employ the coefficients defined for the RGB to YC_bC_r conversion which is part of the JPEG File Interchange Format [155]:

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.1687 & -0.3313 & 0.5 \\ 0.5 & -0.4187 & -0.0813 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}. \tag{5.2}$$

5.2.1 Approximations and Design Space

As mentioned above, the DFG of this application contains arithmetic operations only. It is implemented in scalable fixed-point arithmetic. To approximate the calculations, we combine precision scaling for internal signals together with a choice of approximate multipliers and adders.

In terms of precision scaling, we independently vary the fractional bitwidth F_{co} of each coefficient between 0 and 13, with 13 being the fractional width used in the reference implementation. Furthermore, we scale the precision of the intermediate results after the multiplication. As these signals feed into the subsequent adder chain that expects equally sized inputs, a common parameter F_{int} is used whose range depends on the maximum bitwidth among the coefficients.

The channel mixer DFG contains three multipliers and two adders. For the multipliers, we include the accurate implementation, denoted as *Acc*, and the BAM, which outperformed the other approximate alternatives in the comparison in Section 2.3.4. The BAM exposes the *HBL* and *VBL* parameters whose ranges are restricted by the multiplier input sizes (see Section 2.2.3.1). The HBL is limited to the size of the smaller input which defines the number of partial product rows. Secondly, the VBL cannot be higher than the multiplier's output bitwidth, which is given by the sum of its input bitwidths and translates to the width of the partial product array. At the lower end, the VBL should be at least as high as the HBL to exclude redundant configurations. To restrict the size of the design space further, we removed configurations that lead to very high errors by restricting the upper HBL and VBL range boundaries to half their theoretical maximum value.

For the adders, we select the MA and the LSA based on the comparison in Section 2.3.3, together with the accurate adder. Both approximate adders expose a *split* parameter whose range depends on the intermediate fractional bitwidth F_{int}. In case the LSA was chosen, an additional *select* parameter determines which input is forwarded to the output in the lower part. An overview of all parameters, their notation and ranges is given in Table 5.1. The table also mirrors the order in which the parameterization needs be performed to account for the parameter dependencies.

Table 5.1: Approximation parameters for channel mixer

Name	Notation and Range	Instances
Fractional width of coefficients	$F_{\text{co}}(i) \in [0, 13]$	$i = \{1, 2, 3\}$
Multiplier type	$M_{\text{t}}(i) \in \{\texttt{Acc}, \texttt{BAM}\}$	$i = \{1, 2, 3\}$
BAM HBL[1]	$M_{\text{h}}(i) \in \begin{cases} \left[0, \frac{\text{maxHBL}(i)}{2}\right], & \text{if } M_{\text{t}}(j) = \texttt{BAM} \\ \textit{ignored}, & \text{otherwise} \end{cases}$	$i = \{1, 2, 3\}$
BAM VBL[2]	$M_{\text{v}}(i) \in \begin{cases} \left[M_{\text{h}}(i), \frac{\text{maxVBL}(i)}{2}\right], & \text{if } M_{\text{t}}(j) = \texttt{BAM} \\ \textit{ignored}, & \text{otherwise} \end{cases}$	$i = \{1, 2, 3\}$
Fractional width of intermediate results	$F_{\text{int}} \in \left[0, \max(F_{\text{co}})\right]$	unique
Adder type	$A_{\text{t}}(j) \in \{\texttt{Acc}, \texttt{MA}, \texttt{LSA}\}$	$j = \{1, 2\}$
Adder split point	$A_{\text{p}}(j) \in \left[0, \left(12 + F_{\text{int}}\right)\right]$	$j = \{1, 2\}$
LSA input select	$A_{\text{s}}(j) \in \begin{cases} \{0, 1\}, & \text{if } A_{\text{t}}(j) = \texttt{LSA} \\ \textit{ignored}, & \text{otherwise} \end{cases}$	$j = \{1, 2\}$

[1] maxHBL is given by the size of the smaller input of multiplier i
[2] maxVBL is given by the sum of the input sizes of multiplier i

Design Space Complexity The overall number of potential parameter combinations defines the complexity of the design space. Considering the ranges defined for the parameters listed in Table 5.1, respecting all constraints and parameter dependencies, the full design space of a channel mixer employing precision scaling and the selected choice of approximate arithmetic units in combination contains between 4.225×10^{12} and 4.808×10^{12} possible configurations. The numbers differ slightly depending on the signedness of the matrix coefficients related to the respective mixer. Across all three channel mixers, this number scales further to $D_{\text{mat, full}} \approx 9.766 \times 10^{37}$ combinations in total. If only approximate arithmetic units are considered, the design space significantly shrinks to $D_{\text{mat, arith}} \approx 4.534 \times 10^{28}$ possible configurations. Similarly, for the precision scaling design space, $D_{\text{mat, prec}} \approx 2.737 \times 10^{13}$ configurations are possible.

5.2.2 Genetic Encoding and Operations

As described in Section 4.2.2, the proposed approach uses genes with real-value encoding. Each channel mixer configuration is given by the parameter list

$$K_{\text{mix}} = \left[F_{\text{co}}, M_{\text{t}}, M_{\text{h}}, M_{\text{v}}, F_{\text{in}}, A_{\text{t}}, A_{\text{p}}, A_{\text{s}}\right], \tag{5.3}$$

and the configuration of the complete application DFG is encoded by the concatenation of all 3 individual channel mixer sub-lists:

$$K_{\text{mat}} = \left[K^1_{\text{mix}}, K^2_{\text{mix}}, K^3_{\text{mix}}\right]. \tag{5.4}$$

Mutation and Crossover In the mutation operation, one of the three channel mixer sub-lists is randomly chosen. Then, that channel is mutated by means of full or partial random re-generation, each with equal probability. When the partial re-generation is selected, the fractional coefficient widths F_{co} and the multiplier configurations $M_{\text{t,h,v}}$ remain unchanged while the rest of the mixer sub-list is newly generated.

A single-point crossover is used to combine the configurations between two parents. Due to the parameter dependencies within the channel mixer sub-lists, the crossover point must lie between the individual sub-lists. Consequentially, the crossover will mix the sub-lists from both parents to create offspring.

5.2.3 Optimization Setup

To parameterize the setup of the power model for this target application, we used the simulation model of the approximation-less reference application to estimate internal toggle rates as proposed in Section 3.3.2 using the images from the ARRI image set [134]. The results range between 0.25 and 0.33 among the different images, averaging at 0.3. To extract the application-specific characteristic per-unit power consumption, we synthesized the reference implementation plus four random approximate configurations and averaged the results.

In the target application, the luminance and color information of the input image are separated. The resulting output is typically not meant for display and direct consumption by humans but instead serves as basis for more complex image processing tasks such as image interpretation. Therefore, we choose the PSNR, which is a simple, widely used assessment of the overall accuracy of image processing tasks and systems, for this first case study. In contrast, the other case studies,

which target applications whose output is directly consumed by humans, will feature the ΔE measure, which relates to visual color perception. For the quality training data, since the PSNR relates to the average quality performance across different inputs, we extracted a *real-world data set* consisting of 64^3 randomly sampled inputs from the *Color Wheel* image in *Rec709* encoding of the ARRI image set (cf. Section 3.4.3).

The DSE is run with the two objectives *minimize*(power) and *maximize*(PSNR). As stop condition, we end the GA loop after 750 generations. The usable range in terms of application quality of the RGB to YC_bC_r conversion may vary lightly depending on the further use of the output. For this case study, we set the minimum acceptable quality threshold to a PSNR value of 30dB, which is a lower acceptable level in many image processing applications [18]. The boundaries of the ROI used within the ROI-NSGA selection step of the GA loop to focus the search are consequentially set to a minimum PSNR of 30dB in the quality dimension and to a maximum power of 26.21 mW, which is the power consumption of the reference implementation, as estimated by the proposed power model.

5.2.4 DSE Results

Because of the non-deterministic nature of the employed GA-based optimization procedure, each run of the DSE produces a slightly different front of Pareto-optimal results. Note that the use of the term Pareto-optimal in this context refers to Pareto-optimality among all solutions visited during the same DSE instead of global Pareto-optimality within the entire design space. Unfortunately, due to the excessive size of the design space, it is infeasible to exhaustively extract the ground truth Pareto front with globally optimal solutions against which the results could be compared. Instead, we will assess the solutions based on their practical utility to explore the quality-resource trade-off in real-world implementations.

Figure 5.2 depicts the aggregated results across 10 randomly selected runs using the full design space. The depicted set of solutions offers a multitude of implementation choices across a wide range of different quality-resource trade-offs. Out of these 10 runs, one was randomly selected and is highlighted in the graph.

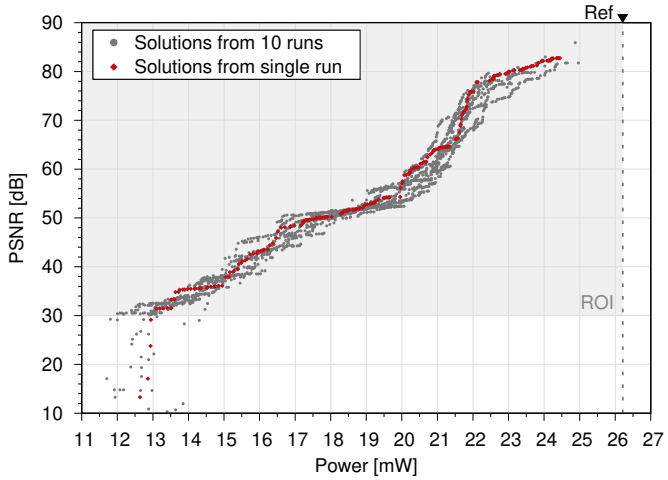


Figure 5.2: Resulting solutions aggregated from 10 independent DSE runs for the full design space of CS1. The results from one randomly selected run are specifically highlighted.

The plot shows that the selected front follows the same overall shape as the aggregated set of solutions, but in many places, slightly better results are contained in the other fronts. Consequentially, we conclude that by repeating the DSE a few times, the amount, density and quality of available solutions can be improved. In the following, we will therefore always present and analyze the results aggregated across 10 randomly selected DSE runs for any of the case studies. Nevertheless, for a more comprehensive check of consistency, we inspected the results from all 30 runs for each experimental setup, observing no significant deviations between different runs with the same setup.

Furthermore, we compare the results of running the DSE with the full design space, i.e. enabling the combination of precision scaling and approximate arithmetic units, with the respective sub-sets contained in the approximate units design space and the precision scaling design space. To that end, Figure 5.3 plots the results from 10 randomly selected runs for each of the three alternative design spaces. Across all design spaces and individual DSE runs, the search was able to focus the vast majority of the results into the ROI and generates well populated fronts from which solutions can be chosen.

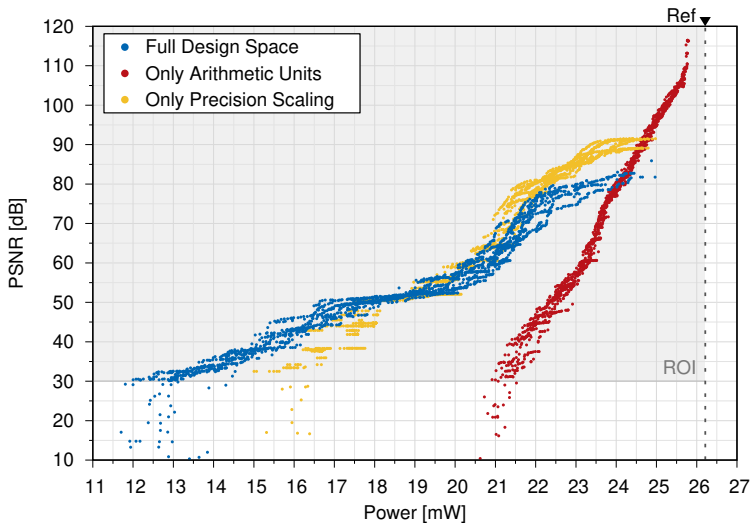


Figure 5.3: DSE results for CS1 obtained for different design spaces, i.e. with all approximations enabled (blue), using only precision scaling (yellow) and using only approximate arithmetic units (red)

Furthermore, it can be seen that the results from the different design spaces dominate different portions of the objective space. The arithmetic units design space yields solutions with the highest quality, dominating the region above a PSNR of 90 dB without competition. However, only moderate power savings (less than 7%) can be reached in this region compared to the reference implementation. At quality levels below 90 dB, the solutions are dominated by the results from the other two design spaces which unlock significantly larger energy savings.

The results from the full design space and precision scaling design space follow similar shapes but outperform each other at the opposing ends of the quality spectrum. For quality levels above a PSNR of approximately 70 dB, the results from using only precision scaling dominate the ones obtained with the full design space. At PSNR values between 50 dB and 70 dB, both design spaces yield similar quality-resource trade-offs, before the results from the precision scaling design space fall off and are dominated by the results of the full design space at quality levels associated with a PSNR below 50 dB.

These results show that in this case study the exploitation of the quality-resource trade-off is quite limited when only arithmetic operations are swapped out for approximate units. Using precision scaling, in contrast, delivers a larger range of trade-off opportunities. This may be due to that fact that precision scaling not only leads to a reduction in the size of operations, but also significantly reduces the number of registers needed for pipelining internal signals as well as the overall routing complexity within the FPGA. These effects are not considered in related works that are limited to the replacement of approximate units and target only purely combinatorial circuits [124, 151, 152]. Nevertheless, only symbiotic combinations of precision scaling and approximate arithmetic units as found within the full design space are able to unlock the largest resource savings while still maintaining PSNR values above 30 dB.

Depending on the further use of the YC_bC_r data, the designer may have a more specific quality level in mind. Across a wide range of PSNR values up to 70 dB, using the full design space is recommended to optimally explore the quality-resource trade-off space. However, when even higher quality levels are desired, a restriction of the design space can lead to better results. However, those solutions should ideally also be found using the full design space, which indicates a potential for further improvements in the optimization methodology.

5.2.5 Model Validation

In the course of a single run of the DSE with the used setup, roughly 7.5×10^4 candidate solutions are probed, which is by many orders of magnitude lower than any of the different design space sizes reported in Section 5.2.1, but still large enough to prevent a fitness estimation using traditional means, especially complete synthesis, fit and route with subsequent placement-aware power analysis. To enable a fast fitness estimation, the proposed models are intentionally simple, using a divide-and-conquer area and power model as well as bit-exact software simulation with a relatively small quality training set. The model-based evaluation of a single solution from the full design space, which includes the bit-exact simulation of approximated arithmetic operations, takes about 0.16 s on the employed workstation computer. In contrast, completing the actual synthesis, fit & route together with power analysis in the respective vendor tool[1] alone takes over 4 min on the same system. However, due to the abstractions and simplifications used in the proposed models, their accuracy as well as their suitability for guiding the DSE needs to be validated. Additionally, the compliance with the target operating frequency needs to be checked.

Therefore, we have selected 12 different solutions from the DSE results obtained with the full design space, which represent a wide range of relevant quality-resource trade-off options, for further analysis. The selected solutions are marked in Figure 5.5a. We synthesized the corresponding configurations and obtained the area and power consumption of the respective designs after synthesis, fitting and routing using the Intel Power Analyzer [137], averaged across 10 different seeds, which will be referred to as *post-synthesis* results.

To validate the quality estimation, we used a set of test images to analyze how well the model results translate to real-world usage of the application. This test set comprises all images of the ARRI image set [134] except for the *Color Wheel* image, which was used to derive the training set. However, in addition to the 11 remaining images, we added a *Corner Case* image to the validation set which is shown in Figure 5.4. It includes content that is often hard to handle in color processing tasks, particularly saturated colored light sources and bright reflections (both white and in a range

[1] In all experiments, the Intel Quartus Prime Pro Edition, Version 18.1.2 Build 277 was used [121]

Figure 5.4: *Corner Case* image which includes colored lights as well as white and colored reflections

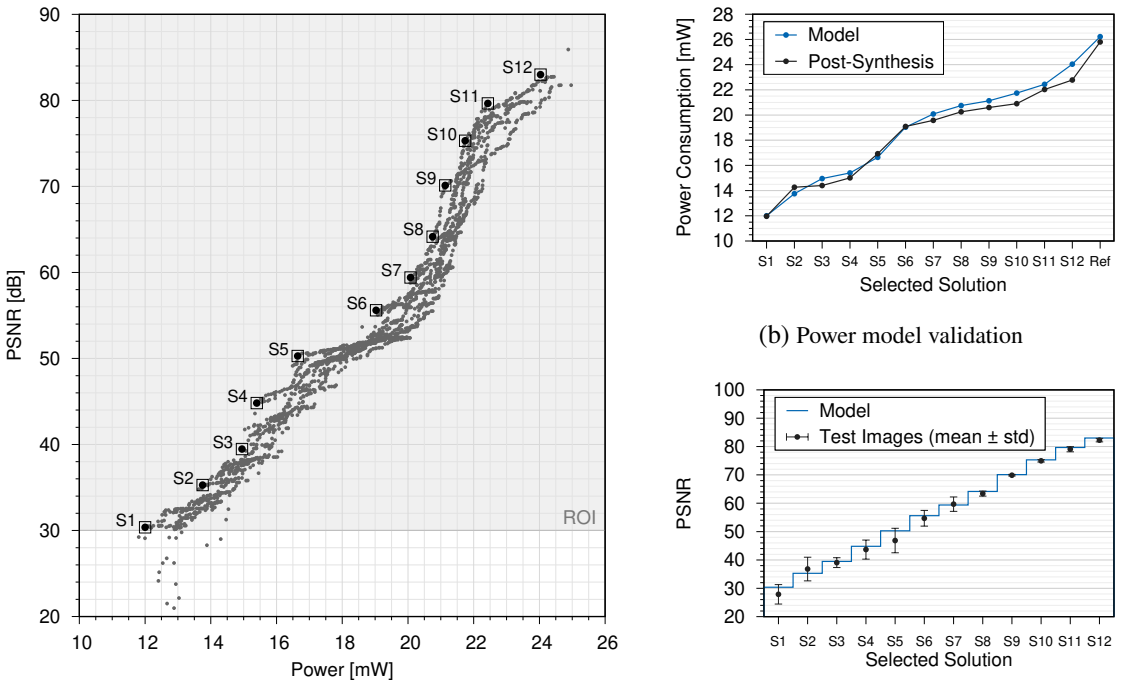
of saturated colors). This addition serves to test the robustness of the estimation. For this case study, we use the validation set in Rec709 encoding which matches the input of the application.

Table 5.2 provides detailed information about the selected solutions. It shows their area and power consumption as estimated by the proposed models and obtained post-synthesis, together with the reported maximum operating frequency. According to the data, the selected solutions reduce the post-synthesis power consumption compared to the reference design by 11% to 54%. Additionally, the table reports the quality estimated with the training set as well as the average PSNR obtained across all test images. In the following, we will analyze the resource, speed and quality properties of the selected solutions and evaluate the accuracy and suitability of the models.

Area and Power Consumption The reported FPGA resource count, listed in Table 5.2, shows that the area model yields highly accurate estimates across all selected solutions. While the number of DSP units is estimated correctly in all cases, the maximum error in LUT count is 4.91% (for S4) with an average deviation of 2.20% across all solutions and the worst estimation of register count (at S2) deviates by only 0.37% from the post-synthesis result, with a mean deviation of 0.23% across all solutions. The comparison of modeled and post-synthesis power consumption is additionally visualized in Figure 5.5b. It can be seen that the model follows the same overall trend that is also observed in the post-synthesis results. The mean deviation of modeled values from the post-synthesis results is 2.47% with the worst-case error observed for S12 at 5.21%.

While the estimations obtained with the proposed resource models differ slightly from the final post-synthesis results, the errors are reasonably low. During the optimization, the most important aspect is that the model follows the correct relation when comparing different solutions, which is confirmed by the reported data. The resource models are therefore capable of guiding the search and give a reliable estimation of the actual resource consumption.

Speed In the proposed framework, all solutions use the same internal pipelining structure as the reference implementation that operates at a speed of up to 272.16 MHz, which is slightly higher than the target frequency of 266.66 MHz. Table 5.2 shows that all of the selected solutions are meeting the target speed requirements as well. Furthermore, it can be seen that with increasing amounts of approximation, the attainable speed of the application also rises. While improving the speed of the implementation is currently not considered as a target objective beyond the satisfaction of a given timing constraint, the results indicate that significant performance improvements are possible through the use of approximations in pipelined FPGA-based stream processing systems.



(a) Selected solutions within ROI

(b) Power model validation

(c) Quality model validation

Figure 5.5: CS1: Selected solutions for model validation from ROI (a), comparison with post-synthesis power values (b) and quality validation with test images from the validation set (c)

Table 5.2: Area, power and quality data of the selected points for CS1

Sel. Point Index	Area (FPGA Resources) DSPs[1]	LUTs Model	LUTs Synth	Registers Model	Registers Synth	Power [mW] Model	Power [mW] Synth	Speed [MHz] Synth	Quality PSNR [dB] Train	Quality PSNR [dB] Test[2]
S1	0	358	353	485	484	12.00	11.96	371.49	30.37	27.59
S2	0	449	428	532	534	13.75	14.27	344.23	35.28	36.67
S3	0	510	497	565	567	14.95	14.41	323.59	39.47	38.97
S4	0	522	498	584	585	15.40	15.01	327.84	44.81	43.46
S5	0	604	588	607	607	16.65	16.92	303.62	50.28	46.66
S6	2	567	556	652	653	19.03	19.09	295.36	55.60	54.54
S7	5	399	392	640	642	20.08	19.58	294.63	59.41	59.57
S8	5	411	425	672	674	20.75	20.26	289.74	64.16	63.40
S9	7	259	258	670	672	21.13	20.59	279.35	70.11	69.92
S10	7	286	277	689	690	21.74	20.90	272.13	75.30	74.96
S11	7	300	298	721	723	22.43	22.03	278.19	79.65	78.99
S12	9	224	223	744	745	24.03	22.78	272.96	83.00	82.23
Ref	9	276	275	840	842	26.21	25.80	272.16	–	–

[1] The estimation of consumed DSPs by the proposed model matches the post-synthesis results for all systems
[2] Given as the average PSNR across all images of the validation set

Quality Figure 5.5c shows a comparison between the quality estimation using the small, randomly sampled real-world training set and the results obtained with the test images from the validation set. In terms of the validation data, the plot shows the average PSNR across all test images together with error bars indicating the standard deviation between the individual images. The estimated and average PSNR values of all solutions are additionally listed in Table 5.2.

The reported data and the plot show that overall, the estimation of quality translates well to the values seen for the validation images. It can be observed that for solutions with very high quality (S8 and higher), the difference between the individual images of the validation set is noticeably smaller than for the other solutions. In this region, the model also matches the validation data most closely. For the solutions with lower quality, the differences between the validation images increases, and the estimation accuracy slightly diminishes. However, the estimated values stay within the standard deviation of the validation set. We observe the largest absolute deviation at S5, where the model overestimates the average quality by 3.62 dB. Hence, the quality model is a good guide for the DSE, but it is recommended to further validate the quality of suitable candidate solutions before final implementation.

Overall, the prediction of resource consumption and application quality by the proposed models delivers a good indication of the actual post-synthesis area and power consumption and of the quality that will be seen in real images. They are therefore well suited for driving the search and provide a good initial fitness assessment for candidate solutions between which a designer may choose.

5.3 Case Study 2: Display Rendering

The second case study, further denoted as CS2, targets a display rendering application containing multiple processing stages, in which image colors are adapted to specific monitor characteristics in terms of dynamic range, color space and electro-optical transfer function (EOTF). A high-level overview of the application is shown in Figure 5.6.

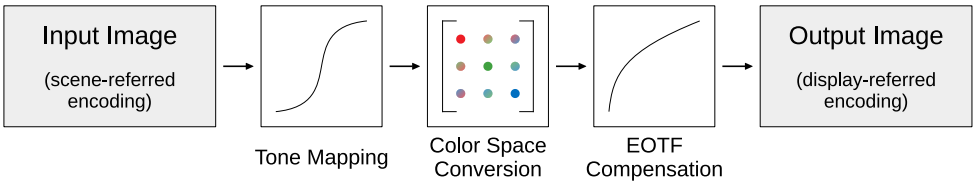


Figure 5.6: Case Study 2: Display rendering pipeline

It can be seen as an extension of the previous case study since the second stage, i.e. the color space conversion, has the same overall structure. However, it serves a different purpose in this context. The additional stages contain non-linear transfer functions which are hard to directly compute in FPGAs. Therefore, such functions are commonly pre-computed and stored in look-up tables on the FPGA device [118]. These stages are predisposed for the integration of table-based approximation techniques which makes it possible to combine three different categories of approximation methods in the same design space, namely precision scaling, approximate arithmetic units and table-based methods. In addition to the full design space that co-integrates all of these approximation types, we also run the DSE with each of the three design sub-spaces corresponding to the restriction to any single type for comparison.

5.3.1 Application Description

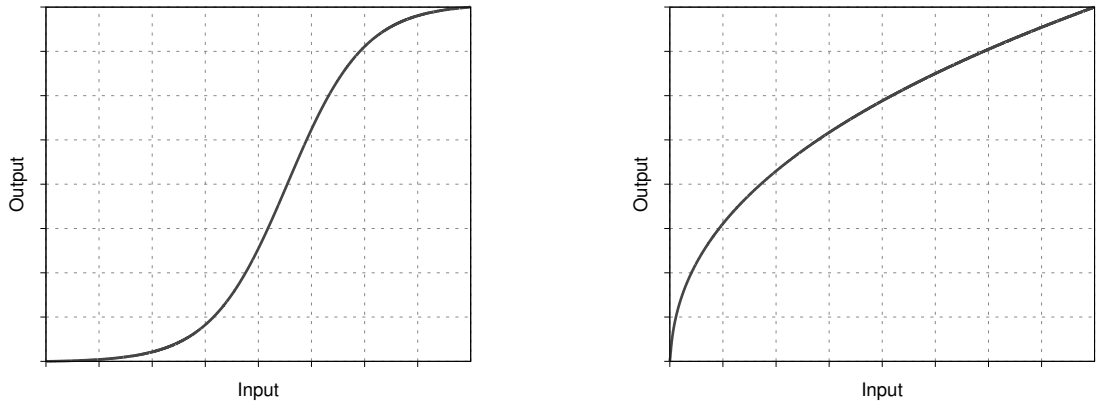
The display rendering pipeline takes image data in a scene-referred encoding which relates to the captured real-world luminances and colors recorded by the camera. In professional motion picture cameras, this typically entails a logarithmic encoding of luminance and a color representation in a so-called wide gamut color space [134]. As mentioned above, three sequential processing stages are then used to adapt the input image to achieve a natural and pleasing rendition on a monitor with specific characteristics. The following sections describe the purpose and functionality of the individual stages in detail and provide the functional parameterization used for our experiments.

5.3.1.1 Tone Mapping

In the first processing stage, a tone mapping operator transforms the input luminance to achieve a natural reproduction of the scene on a display with a different dynamic range. A large variety of different tone mapping techniques can be found in the literature [156]. This case study uses a global sigmoidal function based on a model of photoreceptor behavior [157]. Using this model, the tone mapped values are calculated as

$$X_{\text{tm}} = \frac{\text{enc}^{-1}(X_{in})}{\text{enc}^{-1}(X_{in}) + (hI_a)^k} u + v, \tag{5.5}$$

where $X \in \{R, G, B\}$ is the input/output luminance in any color channel. Since the operator expects linear luminance values, the input values have to be linearized if any non-linear encoding is present in the input data. For this example, we assume a 12b logarithmic encoding which represents the luminance in terms of relative stops between -8 and 8, i.e. $X_{\log} = \frac{X_{\text{in}}}{4095} \cdot 16 - 8$. With this, the logarithmic encoding is reverted using $\text{enc}^{-1}(X_{\text{in}}) = 2^{X_{\log}}$. Furthermore, the variables h, k, and I_a are model parameters which control the overall luminance and contrast of the result. We set them to $h = 9$, $k = 0.6$, and $I_a = 0.4$ in our experiments. Lastly, the values u and v are used to control the numerical range of the output by scaling and shifting the result. In order to map values to the full 12 bit output range, the values are set as $u = 4137$ and $v = -7.4797$. The resulting parameterized transfer function is plotted in Figure 5.7a.



(a) Tone mapping (b) EOTF compensation

Figure 5.7: Plots of the transfer functions used in the display rendering pipeline

5.3.1.2 Color Space Conversion

In the next stage, the colors are converted into the target color space of the monitor using a 3×3 conversion matrix. The values of the matrix depend on the primaries and the white point of the source and target color spaces [158]. This case study employs the values defined for the conversion from Alexa Wide Gamut (AWG) to sRGB [134][2]:

$$\begin{bmatrix} R_{\text{out}} \\ G_{\text{out}} \\ B_{\text{out}} \end{bmatrix} = \begin{bmatrix} 1.4850 & -0.4012 & -0.0838 \\ -0.0337 & 1.2829 & -0.2492 \\ -0.0108 & -0.1220 & 1.1112 \end{bmatrix} \begin{bmatrix} R_{\text{in}} \\ G_{\text{in}} \\ B_{\text{in}} \end{bmatrix}. \tag{5.6}$$

5.3.1.3 EOTF Compensation

In the monitor, digital luminance signals are converted to analog optical light emission. This conversion typically follows non-linear behavior and is characterized by the monitor's electro-optical transfer function (EOTF). To control the light emission across different monitor types, the EOTF needs to be equalized by applying its inverse function. Therefore, the EOTF compensation is the final stage in the display rendering pipeline. Various standards and recommendations define corresponding transfer functions for different devices. In this work, we use the function defined in the sRGB standard [159]:

$$X_{out} = \begin{cases} 12.92\, X_{in} & \text{, for } X_{in} < 0.0031308, \\ 1.055\, X_{in}^{1/2.4} - 0.055 & \text{, for } X_{in} \geq 0.0031308, \end{cases} \tag{5.7}$$

where $X \in \{R, G, B\}$ represents any color channel. The resulting transfer function is plotted in Figure 5.7b.

5.3.1.4 High-Level Application Structure

The high-level structure of the display rendering application is depicted in Figure 5.8. It shows the channel-wise application of the transfer functions in the tone mapping and EOTF compensation stages together with channel mixers in the color space conversion stage. In the transition between stages, the signals will be fixed to 12 bits to break potential parameter dependency paths across stage borders. The fixation of these interfaces therefore ensures that DFG nodes and their parameters can be grouped meaningfully which simplifies the definition of genetic operations and enables reusing the operations defined for CS1 in the color space conversion stage.

5.3.2 Approximations and Design Space

Even though it serves a different purpose in the display rendering application, the structure and general functionality of the color space conversion stage is the same as in CS1. Hence, this stage employs the same approximations, related parameters and ranges as detailed in Section 5.2.1.

[2] Please note that these values are not the ones reported in the text of the cited paper but rather the values used in the code that was published together with that paper. After reaching out to the authors of the paper, it was confirmed that the values reported here are the correct ones.

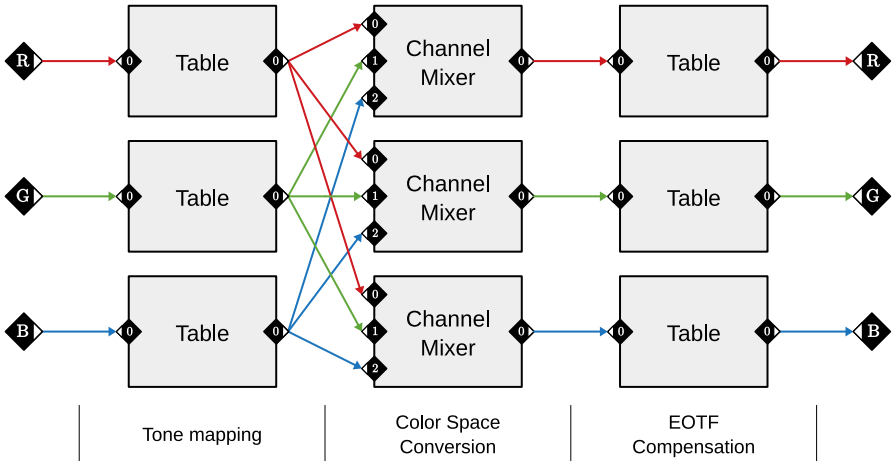


Figure 5.8: High-Level Structure of the display rendering application

However, this case study augments the design space of CS1 by adding table-based approximations to reduce the memory consumption in the other two stages. For this, sparse tables with the hierarchical segmentation scheme proposed by Lee et al. [87] as described in Section 2.2.4 are used. First, the input range is split into N_{sec} major sections, which is limited to less than or equal to 16 in our experiments. In the second level, each of these sections is divided further into $N_{\text{seg}}(i)$ sub-segments, where i indicates the section index. The upper limit of this parameter is naturally given by the input range covered by a section, i.e. the overall input range divided by the number of sections. Furthermore, we add the constraint that the total sub-segment count across all sections must be at least 16 to remove configurations of impractically low quality. Both N_{sec} and $N_{\text{seg}}(i)$ are limited to powers of two which enables address mapping with little combinatorial overhead. To reduce the reconstruction error between the sparse grid points, linear interpolation can optionally be enabled, as controlled by the parameter I. A summary of these parameters and their ranges is given in Table 5.3. Since the same transfer function is applied on all three channels in the respective processing stages, all three parallel instances share the same parameterization.

Table 5.3: Approximation parameters for the sparse tables

Name	Notation and Range	Instances
Interpolation mode	$I \in \{\texttt{NONE}, \texttt{LINEAR}\}$	unique
No. of sections	$N_{\text{sec}} \in \{1, 2, 4, 8, 16\}$	unique
No. of sub-segments	$N_{\text{seg}}(i) \in \left\{1, 2, 4, .., \frac{2^{12}}{N_{sec}}\right\}$, so that $\sum_{i=1}^{N_{\text{sec}}} N_{\text{seg}} \geq 16$	$i = [1, N_{sec}]$

Design Space Complexity The number of parameters used to configure a sparse table depends on the number of sections N_{sec}. Therefore, the overall design space complexity of the related design space is calculated by summing up the possible combinations across the vector of sub-segment counts N_{seg} and the interpolation mode I. Under consideration of the parameter ranges given in Table 5.3, the design space of the tone mapping respectively the EOTF compensa-

tion step, which both employ a sparse table approximation, contains $D_{\text{sparse}} \approx 3.706{\times}10^{15}$ possible configurations. The parameters and ranges for the channel mixers of the color space conversion matrix are the same as defined in Table 5.1 for CS1. Overall, considering all possible parameter combinations across the entire application and its three processing stages, the complexities of the different design spaces amount to the following values:

- **Full design space:** $D_{\text{disp, full}} \approx 1.526 \times 10^{69}$
- **Sparse tables design space:** $D_{\text{disp, table}} \approx 1.373 \times 10^{31}$
- **Arithmetic units design space:** $D_{\text{disp, arith}} \approx 5.439 \times 10^{28}$
- **Precision scaling design space:** $D_{\text{disp, prec}} \approx 2.737 \times 10^{13}$

5.3.3 Genetic Encoding and Operations

The genetic encoding of the configuration for the color space conversion matrix follows the same scheme as defined in Section 5.2.2 for CS1, yielding the list K_{mat}. Similarly, the configuration of a sparse table is represented by a list of the parameters described in Table 5.3 in order of their parameterization:

$$K_{\text{sparse}} = \left[I, N_{\text{sec}}, N_{\text{seg}}\right]. \tag{5.8}$$

Since the bitwidth in the interface between the major pipeline steps of CS2 is fixed to 12 bits, any of the steps can safely be configured independently. Therefore, the overall configuration of the entire display rendering application can be represented by concatenating the individual encodings for the tone mapping step, the color space conversion step and the EOTF compensation step into a larger list:

$$K_{\text{disp}} = \left[K^{\text{TM}}_{\text{sparse}}, K_{\text{mat}}, K^{\text{EOTF}}_{\text{sparse}}\right]. \tag{5.9}$$

When evolving offspring from parents, mutation and crossover are generally executed independently for each of the three sub-lists that represent the different processing stages. In the mutation operation, one pipeline step is chosen at random, and the mutation operation related to that step is performed, modifying only the respective sub-list of the genetic encoding. Contrarily, in the crossover operation, all steps and the respective sub-lists are affected, but each step is modified separately, using the crossover operation related to it. This approach allows for changes and extensions in the target application, e.g. removing, replacing or adding a major processing stage to the pipeline, without the need to introduce changes to the GA implementation.

For the matrix multiplication in the color space conversion stage, the same genetic operations as defined in Section 5.2.2 are used. The operations performed for the sparse tables used for the transfer functions in the tone mapping and the EOTF compensation stages are defined as follows:

Mutation Similar to the mutation of the channel mixer encoding, the algorithm chooses randomly, with equal probability, between a full and a partial random re-configuration. In the partial reconfiguration, the interpolation mode I and the number of sections N_{sec} is carried over and the vector N_{seg} containing the number of sub-segments per section is randomly re-generated.

Crossover The N_{seg} vectors may have different lengths between the two parents chosen for the crossover operations, depending on their respective values for N_{sec}. Hence, a direct single-point crossover cannot be safely used. To circumvent this issue, the crossover operation needs to ensure the vectors are of equal length. Therefore, we implemented mechanisms to *up-sample* or *down-sample* one of the parents so that both have the same encoding length. Consider two parents with $N_{\text{sec},1} < N_{\text{sec},2}$ and the ratio $r = N_{\text{sec},2}/N_{\text{sec},1}$ that needs to be equalized. In up-sampling, the number of sections $N_{\text{sec},1}$ is scaled up by r so that it matches the second parent, and each entry $N_{\text{seg},1}(i)$ is replaced by r scaled copies of $N_{\text{seg},1}(i)/r$. The overall number of segments and the segment distribution stay the same when up-scaling, except for the case that the resulting entries would be below 1. In that case, they are directly set to 1, which slightly increases the overall number of segments. On the contrary, when down-sampling the other parent, $N_{\text{sec},2}$ is scaled down by $1/r$ to match the first parent, and each group of r consecutive entries of $N_{\text{seg},2}$ is replaced by its sum, rounded to the nearest power of two to respect the constraints of the sparse table architecture. Note that the rounding step can lead to a change in the overall number of segments. As an example, the down- respectively up-sampling of a parent with $N_{\text{seg}} = [256, 512]$ works as follows:

$$[1024] \xleftarrow{\text{down-sample}} [256, 512] \xrightarrow{\text{up-sample}} [128, 128, 256, 256]$$

The crossover operation randomly chooses between up-sampling or down-sampling with equal probability so that the evolution is not biased towards either smaller or larger values of N_{sec}. Once the encoding of both parents has been adapted to the same length, a single-point crossover can be performed, splitting the parents within the N_{seg} vector.

5.3.4 Optimization Setup

The internal toggle rates range between 0.22 and 0.33 between different images from the ARRI image set [134]. Their average is roughly 0.28, which is what we set as toggle rate for the experiments. For the application-specific characteristic per-unit power consumption, we extracted data from the reference implementation as well as 3 approximate configurations that we deliberately selected out of 10 random configurations so that different amounts of resource consumption are covered, and averaged the results.

This application renders images for display, hence its output is directed for human consumption. In order to reliably judge the application quality, we therefore use the ΔE color accuracy measure [139]. As main target, we want to bound the worst case color deficiency, i.e. to minimize the maximum ΔE across the color range. However, in preliminary experiments we found that setting the maximum error as the only quality target in GA-based optimization leads to a less efficient search. This is likely because in that case the optimization neglects small parameterization changes that lead to improvements in the average error but are not reflected in the worst-case error. However, in the long run, a sequence of such changes may eventually culminate in a reduction of the worst-case error as well. Therefore, we added the mean ΔE as a secondary quality objective, which improved the optimization performance. As training set, we chose a synthetically generated data set by regularly sampling the input color space and adding noise as proposed in Section 3.4.3 since the main objective is to bound the worst-case color difference. For the experiments, we used 128 sampling steps in each direction which results in 128^3 input points in the training set.

With both selected quality targets, the DSE is set up with three objectives overall, namely to *minimize*(power), *minimize*(max ΔE) and *minimize*(mean ΔE). Each experiment is run until 1000

generations have evolved. Experimental studies by Stokes et al. applied differently scaled disturbances to a set of test images, which lead to different color differences compared to the original image [160]. Their experiments resulted in median visual perceptibility thresholds between 1.43 and 2.63, depending on the type of disturbance but without significant impact of image content, resulting in an average threshold of $\Delta E = 2.15$. This gives a rough indication about the range in which color differences may be acceptable. For the experiments, we set the ROI boundaries to $\max \Delta E = 5$, $\text{mean}\,\Delta E = 2.15$ and power $= 57.82\,\text{mW}$ (which is the power estimation of the reference implementation), to focus the DSE search towards solutions relevant for practical implementation.

5.3.5 DSE Results

Because we defined three target objectives for the optimization in this case study, the DSE yields a three-dimensional trade-off front. However, the reason to add the minimization of the mean ΔE value as additional objective was mainly to improve the optimization performance. Hence, the most relevant trade-off is given between the main targets, the maximum ΔE and the power consumption. Figure 5.9 therefore depicts a projection of the results onto these two objectives. Similar to CS1, we randomly selected 10 runs from each of the four design spaces and plotted the aggregated solutions in the figure. Solutions with a maximum ΔE above 15 have been excluded from the plot to improve visibility.

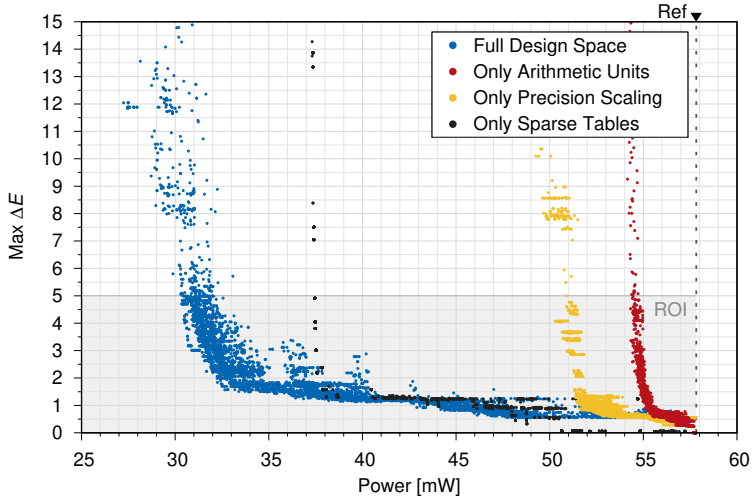


Figure 5.9: DSE results for CS2 obtained for different design spaces, i.e. with all approximations enabled (blue), using only approximate arithmetic units (red), using only precision scaling (yellow) and using only sparse tables (black)

Comparing the different design spaces, the plot shows that the full design space offers the widest range of relevant quality-resource trade-offs to choose from, with a densely populated front covering estimated power savings up to almost 50% within the ROI. In contrast, the reduced sub-spaces are falling off beyond the quality boundary at higher power levels. The smallest benefits can be achieved with the arithmetic units design space, which enables reasonable options down to an estimated power consumption of approximately 54 mW, which is only a small reduction of less than

7% from the reference power of 57.82 mW. Next, the precision scaling design space roughly doubles the savings, widening the relevant trade-off range down to around 50 to 52 mW of estimated power. The sparse tables design space is the most proficient between the reduced ones, enabling power savings of up to 35% while keeping the maximumΔE below 5. Furthermore, towards the high-power/high-quality range at power values above 50 mW, it outperforms the results from all competing design spaces by finding solutions with the lowest errors overall. However, moving in direction of more resource savings, the resulting fronts become less populated.

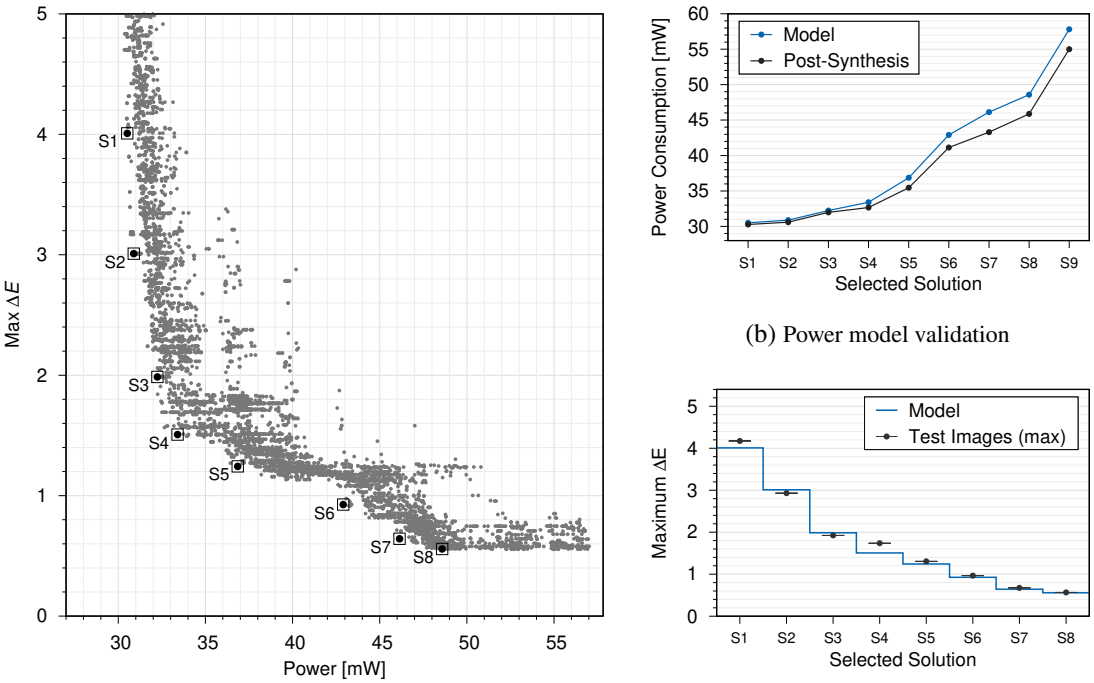
Similar to the results from CS1, this analysis shows that the simultaneous combination of different approximations offers the largest range of potential quality-resource trade-offs, but in cases where the highest possible quality is needed, marginally better options are found in the sparse table design space. Furthermore, the different fall-off points of the reduced design spaces indicate that the power consumption of this application is dominated by the embedded memory in terms of BRAMs used to implement the transfer functions, since across the reduced design sub-spaces, the most benefits are achieved when using the sparse table approximations.

5.3.6 Model Validation

To validate the accuracy of the model-based resource and quality prediction for CS2, 8 solutions were selected from the results obtained with the full design space, covering a wide range of quality-resource trade-off options across the ROI. The selected solutions are highlighted in Figure 5.10a. Numerical validation data for each of these solutions as well as the reference implementation is reported in Table 5.4. The post-synthesis power consumption is reduced by the selected solutions between 16% and 45% in comparison to the reference implementation. In terms of the quality validation, the table lists the maximum ΔE as estimated using the synthetically sampled training data set together with the maximum ΔE observed across all images of the validation set, for which the same images were used as for CS1, including the *Corner Case* image (see Figure 5.4). However, since this application expects input data in scene-referred encoding, the *logC* versions of the images were used as validation set.

Area and Power Consumption Since the number of DSPs and the amount of memory implemented using BRAMs can directly be derived without error from the configuration, the consumption of these units are always predicted correctly. In terms of LUTs and registers, the estimation errors are very low, with maximum deviations of 4.84% for LUTs (at S4) respectively 0.37% for registers (at S8) and an average deviation of 2.78% for LUTs and 0.24% for registers. Figure 5.10b visualizes the comparison of modeled and post-synthesis power values, showing that the model captures the overall trend closely. However, it can be noticed that it tends to overestimate the power values, especially within the high-power region. Nevertheless, the worst-case deviation of 6.15% (at S7) and the average deviation of 3.25% are quite low. Overall, the proposed models deliver a reliable estimation of the area and power consumption.

Speed With regards to the achievable speed, a similar trend can be observed as for CS1. All of the solutions meet or exceed the specified target operating frequency of 266.66 MHz, with speed trending towards higher values with increasing amounts of approximation. However, in CS2 this trend is less pronounced, and the achievable speed improvements are overall more limited compared to the results of CS1.



(a) Selected solutions within ROI

(b) Power model validation

(c) Quality model validation

Figure 5.10: CS2: Selected solutions for model validation from ROI (a), comparison with post-synthesis power values (b) and quality validation with test images from the validation set (c)

Table 5.4: Area, power and quality data of the selected points for CS2

Sel. Point Index	FPGA Resources						Power		Speed	Quality	
	DSPs[1]	BRAMs[1] [Units/Bits]	LUTs		Registers		[mW]		[MHz]	max ΔE	
			Model	Synth	Model	Synth	Model	Synth	Synth	Train	Test[2]
S1	2	6 / 73 728	623	615	826	828	30.51	30.28	292.49	4.01	4.17
S2	2	6 / 73 728	630	622	847	848	30.89	30.60	294.38	3.01	2.93
S3	2	6 / 73 728	734	714	866	868	32.24	31.98	292.62	1.99	1.93
S4	6	6 / 69 120	413	434	889	887	33.40	32.67	272.97	1.51	1.74
S5	5	6 / 92 160	641	649	1036	1038	36.85	35.45	274.33	1.24	1.31
S6	5	12 / 184 320	471	490	884	883	42.90	41.12	272.74	0.93	0.96
S7	6	12 / 202 752	480	494	1029	1026	46.13	43.30	273.25	0.64	0.67
S8	7	12 / 199 680	560	541	1069	1073	48.58	45.88	272.71	0.56	0.57
Ref	9	18 / 294 912	292	302	1064	1068	57.82	55.01	269.83	–	–

[1] The estimation of consumed DSPs and BRAMs by the proposed model matches the post-synthesis results for all systems

[2] Given as the overall maximum ΔE observed across all images in the validation set

Quality The comparison of modeled worst-case color difference and the actual maximum ΔE observed across the test set is given in Table 5.4 and additionally visualized in Figure 5.10c. It can be seen that while the actual worst-case disturbance found in the real images often slightly exceeds the modeled value, it stays within a reasonable margin of error. The largest deviation is seen at S4, where the estimated maximum ΔE is exceeded by 0.23 in the validation set. In practical situations, the designer may pre-select relevant solutions with modeled worst-case color differences slightly below the actual target and evaluate their worst-case error with a bigger training set, i.e. even more synthetically sampled input colors, a number of test images such as the validation set, or a combination of both, to increase the safety of the error bound.

5.4 Case Study 3: 3D Lookup Table Color Processing

The third case study, denoted as CS3, covers another very common application often used in digital photography or motion picture production. It deals with the transformation of image colors by means of a 3D lookup table (3D-LUT). In principle, it uses the same concept as the use of precalculated data for one-dimensional transfer functions, but extends it to three dimensions. Hence, a 3D-LUT tabulates full target output colors in terms of their RGB components and in turn is addressed by the three-dimensional input color values. Compared to the separate use of a one-dimensional transfer function on each color component, a 3D-LUT allows more degrees of freedom in the calculation, enabling widely variable transformations of different input colors and offering a more nuanced control over brightness, contrast, hue and saturation. 3D-LUTs are commonly used to render input images in scene-referred encoding into a color-graded final image in display-referred encoding, incorporating a desired look into the image. Figure 5.11 depicts a high-level overview of the target application containing a 3D-LUT processing stage.

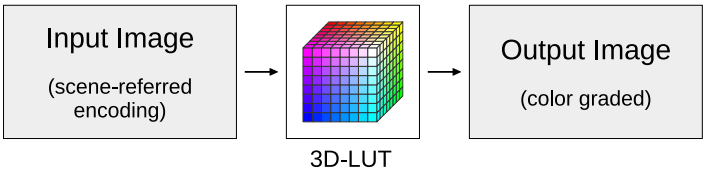


Figure 5.11: Case Study 3: Processing pipeline for 3D-LUT rendering application

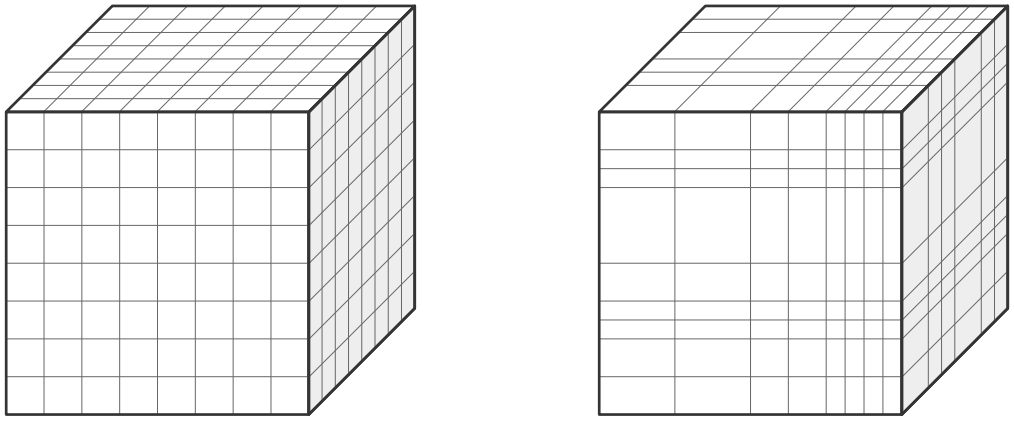
As the size of the input space scales exponentially with each input dimension, it is impractical to store a full lookup table for all possible input colors. Instead, sparse tables with uniform segmentation are typically used and multi-dimensional interpolation is employed to reconstruct the outputs for intermediate input colors [161]. Hence, classical implementations of 3D-LUTs already inherently incorporate approximation by the choice of lattice density, typically using 16 or 32 segments per dimension in hardware systems and often 64 or more in software implementations. In this case study, we incorporate the use of non-uniform hierarchical sparse table segmentation (as used for the one-dimensional transfer functions in the previous case study) together with targeted precision scaling and approximate multiplication in the interpolation step to achieve better quality-resource trade-offs compared to the traditional baseline implementation. For the color transformation itself, we employ the ARRI Rec709 default look, using a high-precision 64-segment 3D-LUT exported from the ARRI Color Tool [162] as baseline reference.

5.4.1 Application Description

The implementation of a 3D-LUT contains three major parts: packing, extraction and interpolation. A detailed description of color transformation using 3D-LUTs is given by Kang [161]. An overview of the employed implementation and its structure as used in our case study is given in the following sections.

5.4.1.1 Packing

In packing, the input space is sampled and a sparse lattice of points to be stored in the table is formed. As mentioned, typical implementations use uniform segmentation, but non-uniform segmentation can enable an optimized output reconstruction [161]. The principle of applying a non-uniform segmentation to the 3D-LUT lattice is illustrated in Figure 5.12b in comparison to traditional uniform segmentation as shown in Figure 5.12a. Once the packing structure is chosen, the desired color transformation is sampled at the lattice points and the outputs are tabulated into memory. Each memory word holds the complete output color, i.e. the 36 bit wide RGB triplet related to the respective lattice point.



(a) Classical uniform 3D-LUT segmentation (b) Exemplary non-uniform 3D-LUT segmentation

Figure 5.12: Comparison of uniform and non-uniform 3D-LUT segmentations for packing

5.4.1.2 Extraction

During runtime, the 3D-LUT is accessed by the input color triplet, and the respective sub-cube that contains the input color needs to be found. First, the lattice point number corresponding to the lower sub-cube boundary is identified for each dimension individually. With uniform segmentation, the lattice number simply corresponds to the $\log_2(n)$ MSBs of the input value in each dimension when using n segments, and the remaining bits form the offset within the cube for interpolation. In the case of a non-uniform segmentation, more calculation is needed. As our implementation employs the same hardware-efficient hierarchical segmentation scheme as used in the previous case study, three instances of the premapper described in Section 2.2.4 are used to calculate the individual lattice numbers and interpolation offsets. The offsets are automatically scaled correctly relative to the respective sub-cube side lengths.

For the employed trilinear interpolation (see Section 5.4.1.3), all 8 vertices of the extracted sub-cube must be retrieved from memory. Generally, the entire 3D-LUT is stored in column-major order, i.e. with the first input dimension mapping to contiguous memory locations. To allow an efficient use of dual-port BRAM (which is commonly available on current FPGA devices), the table is split into 4 distinct tables adhering to an even/odd split in the second and third input dimension. This means that the first table stores all lattice points with even indices in both the second and third dimension, the second table contains all values with odd indices in the second dimension but even indices in the third dimension, and so forth. Using this table architecture ensures that all 8 vertices can always be read simultaneously from dual-port memory without requiring redundant table entries. To calculate the memory address, a common base address is calculated first as

$$a_{\text{base}} \quad = \quad a_0 \quad + \quad (a_1 >> 1) \cdot S_1 \quad + \quad (a_2 >> 1) \cdot S_2, \tag{5.10}$$

where a_0, a_1 and a_2 are the lower lattice indices obtained from the premapper address outputs in the first, second and third input dimension (typically referring to red, green and blue color components, respectively), and S_1 as well as S_2 are the array strides, i.e. the number of table entries between a unit increment in the second respectively third dimension. The LSBs of a_1 and a_2, which are discarded in the calculation of the base address, contain information about the even/odd position of the lower sub-cube vertices and are concatenated into a *select* signal. To calculate the final addresses for each of the 8 sub-cube vertice points, different stride-based address offsets are added to the base address as chosen by the select signal. The 8 resulting addresses are distributed across the dual input ports of the 4 tables to access the table content. Finally, the select signal is used to consistently sort the individual memory outputs to signals which represent the 8 vertices in a defined order, yielding the vertice data points p_{000} through p_{111}. The offsets o_0, o_1 and o_2, which represent the relative position of the input within the sub-cube in each dimension, are directly generated by the respective premapper block and forwarded to the interpolation step. Figure 5.13 illustrates the overall structure of the 3D-LUT extraction procedure.

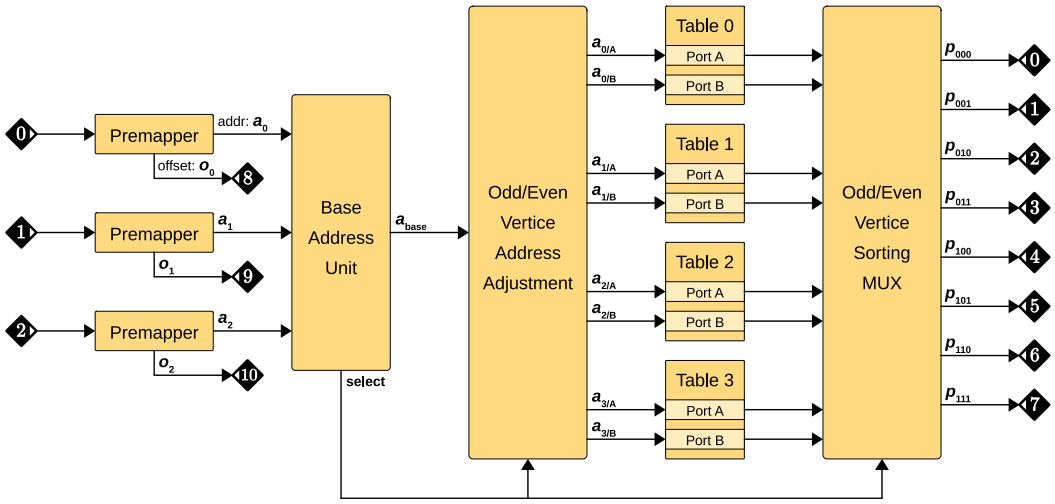


Figure 5.13: Implementation structure of the 3D-LUT extraction procedure

5.4.1.3 Interpolation

Various geometrical interpolation methods have been proposed for output reconstruction at intermediate input values, including trilinear, prism, pyramid and tetrahedral interpolation [161]. For simplicity, our implementation uses a straightforward trilinear interpolation scheme which is instantiated once per output channel. The trilinear interpolation executes 7 linear interpolations in total, interpolating in consecutive stages along each of the three dimensions. First, 4 interpolations are calculated along the respective sub-cube edges of the first dimension using the offset o_0 which represents the relative position of the input on those edges. This first step takes all 8 vertices and yields 4 intermediate values p_{00}, p_{01}, p_{10}, p_{11}. The second stage takes these intermediate values and interpolates along the second dimension, using offset o_1, resulting in two further intermediate values p_0 and p_1. Finally, the last stage interpolates between those two values along the third dimension according to offset o_2 to provide the final output p_{out}. Figure 5.14 depicts the overall structure of the trilinear interpolation.

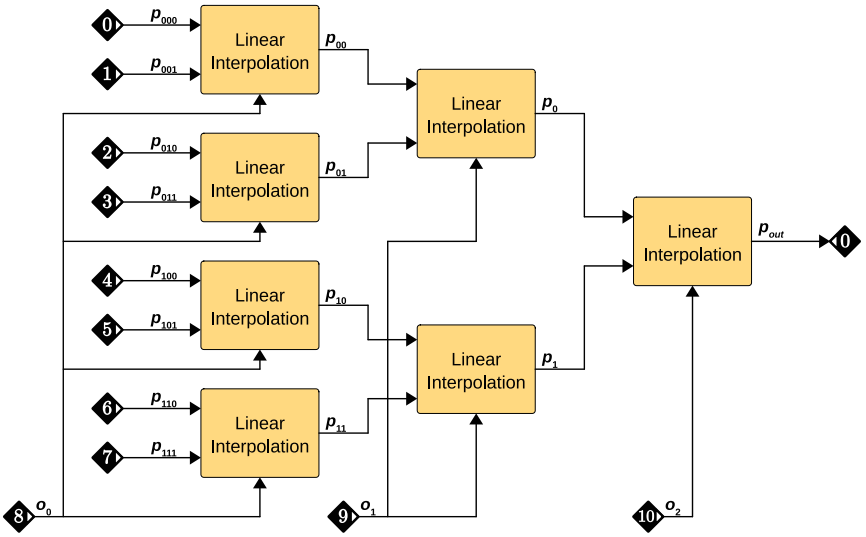


Figure 5.14: Implementation structure of the trilinear interpolation

Each of the linear interpolation blocks calculates its output as

$$q_{\text{out}} = q_0 + (q_1 - q_0) \cdot o \tag{5.11}$$

where the lower and upper vertice values of the given edge are represented by q_0 and q_1, respectively, and the relative position of the input along the edge given by the offset o. The corresponding DFG of the linear interpolation block is shown in Figure 5.15.

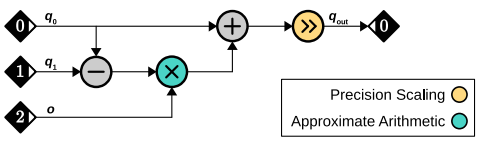


Figure 5.15: Annotated DFG of a single linear interpolation block

5.4.1.4 High-Level Application Structure

Each entry of the 3D-LUT tables stores the entire RGB triplet of the respective destination color in a 3×12 bit word. However, each of the color components needs to be interpolated separately. Hence, the 8 vertice points are split into the respective color components which are then fed into individual trilinear interpolation blocks. Nevertheless, each interpolation block uses the same interpolation offsets o_0, o_1, o_2 for their calculations. Figure 5.16 shows the resulting high-level structure.

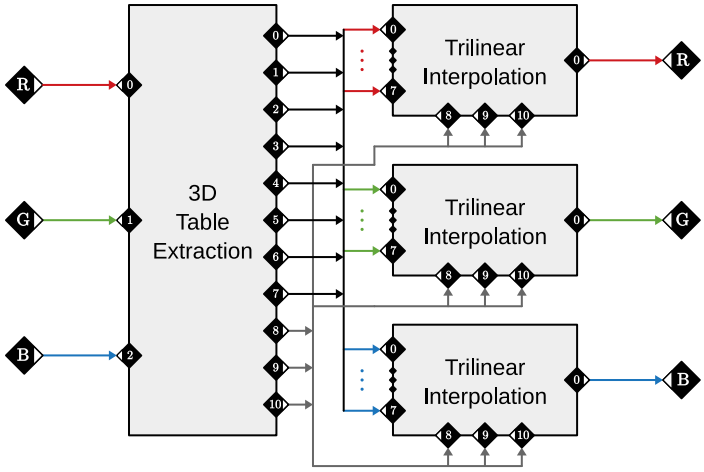


Figure 5.16: High-level structure of the 3D-LUT application

5.4.2 Approximations and Design Space

As mentioned above, standard implementations of 3D-LUT hardware applications already employ a basic form of approximation by storing a three-dimensional sparse table with uniform segmentation, typically using 16 or 32 segments across each dimension. In this case study, we extend this design space by using the hierarchical non-uniform segmentation scheme described in Section 2.2.4 and apply it to each of the three input dimensions of the 3D-LUT. Since the overall table size scales exponentially with each dimension, the parameter ranges are restricted more tightly than in the last case study. In the upper segmentation level, we allow a choice between 4 or 8 sections per channel for N_{sec}. To control the overall number of segments, we restrict their total sum in each dimension into the range between 8 and 48, which allows a redistribution among the input channels at similar overall sizes as when using 16 or 32 in all dimensions. From initial experiments, we have observed that the ratio between the largest and smallest number of segments within each dimension tends to be limited in most of the results. Therefore, we restricted the maximum spread factor between the entries within any N_{seg} vector to 4, which had a positive influence on the optimization performance. The resulting segmentation parameters for a single dimension and all range restrictions are listed in Table 5.5.

Additionally, we add arithmetic approximation and precision scaling in deliberate places within the interpolation procedure. In terms of arithmetics, we enable a choice between the accurate multiplier implementation and the BAM which has been used also in the previous case studies. To simplify the design space, we choose a single multiplier configuration for each interpolation stage, so that the first stage has four equivalent multiplier instances (one per linear interpolation block),

Table 5.5: Approximation parameters for non-uniform hierarchical segmentation in CS3

Name	Notation and Range	Instances
No. of sections	$N_{\text{sec}} \in \{4, 8\}$	unique
No. of sub-segments	$N_{\text{seg}}(i) \in \{1, 2, 4, 8\}$, so that $8 \leq \sum_{i=1}^{N_{\text{sec}}} N_{\text{seg}} \leq 48$ and $\frac{\max(N_{\text{seg}})}{\min(N_{\text{seg}})} <= 4$	$i = [1, N_{sec}]$

the second stage has two identical multipliers and the last stage has one. Also, we did not choose approximate adders as the additions create a negligible amount of resource consumption in this application. However, we additionally scale the width of the intermediate results after each interpolation stage, using a common width parameter $F_{\text{int}}(1)$ for signals p_{00}, p_{01}, p_{10}, p_{11} and another width parameter $F_{\text{int}}(2)$ for the signals p_0 and p_1 resulting from the second interpolation stage. Compared to the precision scaling applied within the channel mixers in the previous steps, this parameter represents not just a fractional width but the entire bitwidth of the intermediate interpolation results. We let these bitwidths range between 10 and 14, with 14 being used in the baseline implementation. Figure 5.15 indicates the related annotations in the DFG of the linear interpolation. The respective parameters for a single trilinear interpolation block are summarized in Table 5.6. Each of the three trilinear interpolation blocks (cf. Figure 5.16) is configured separately.

Table 5.6: Approximation parameters for an individual trilinear interpolation block in CS3

Name	Notation and Range	Instances
Multiplier type	$M_{\text{t}}(i) \in \{\texttt{Acc}, \texttt{BAM}\}$	$i = \{1, 2, 3\}$
BAM HBL[1]	$M_{\text{h}}(i) \in \begin{cases} \left[0, \frac{\text{maxHBL}(i)}{2}\right], & \text{if } M_{\text{t}}(j) = \texttt{BAM} \\ \textit{ignored}, & \text{otherwise} \end{cases}$	$i = \{1, 2, 3\}$
BAM VBL[2]	$M_{\text{v}}(i) \in \begin{cases} \left[M_{\text{h}}(i), \frac{\text{maxVBL}(i)}{2}\right], & \text{if } M_{\text{t}}(j) = \texttt{BAM} \\ \textit{ignored}, & \text{otherwise} \end{cases}$	$i = \{1, 2, 3\}$
Fractional width of intermediate results	$F_{\text{in}}(j) \in [10, 14]$	$j = \{1, 2\}$

[1] maxHBL is given by the size of the smaller input of multiplier i
[2] maxVBL is given by the sum of the input sizes of multiplier i

Design Spaces In this case study, we compare two different design spaces. An analysis of the resource consumption in the baseline system shows that the application is largely dominated by the memory used for the 3D table. Hence, we investigate the potential with a *non-uniform table design space* that is limited to use only the non-uniform segmentation method and compare it to results obtained with the full design space, including approximate multipliers and precision scaling.

Design Space Complexity When restricting the design space to include only the lattice segmentation along each of the three input dimensions, it contains $D_{\text{3D, table}} \approx 2.153 \times 10^{12}$ possible configurations. The extension to the full design space adds a range of interpolation configurations for each parameterization of the table, bringing the complexity up further to a total number of $D_{\text{3D, full}} \approx 5.699 \times 10^{18}$ configurations.

5.4.3 Genetic Encoding and Operations

The genetic encoding of the sparse table parameters is similar to the one used in the previous case study, but without the interpolation parameter because interpolation is always used in this case:

$$K_{\text{sparse}} = \left[N_{\text{sec}},\ N_{\text{seg}}\right]. \tag{5.12}$$

Three instances of this list are used for the extension to three dimensions:

$$K_{\text{3D, table}} = \left[K^0_{\text{sparse}},\ K^1_{\text{sparse}},\ K^2_{\text{sparse}}\right], \tag{5.13}$$

which fully describes the configuration in the restricted non-uniform table design space.

For the full design space, we additionally encode the parameterization of any trilinear interpolation block as

$$K_{\text{trilinear}} = \left[M_{\text{t,h,v}}(1),\ F_{\text{in}}(1),\ M_{\text{t,h,v}}(2),\ F_{\text{in}}(2),\ M_{\text{t,h,v}}(3)\right], \tag{5.14}$$

which matches the correct parameterization order of configuration. Three instances of the trilinear configuration are concatenated to cover the reconstruction of all color components:

$$K_{\text{3D, interp}} = \left[K^1_{\text{trilinear}},\ K^2_{\text{trilinear}},\ K^3_{\text{trilinear}}\right]. \tag{5.15}$$

The configuration of an entire candidate solution in the full design space is then represented by

$$K_{\text{3D, full}} = \left[K_{\text{3D, table}},\ K_{\text{3D, interp}}\right]. \tag{5.16}$$

Mutation and Crossover Similar to the approach used for the separate channel mixer, the three sub-lists in $K_{\text{3D, table}}$, which represent the table segmentation in each respective dimension, are treated independently. The mutation operation chooses one of them randomly and performs either a full or a partial random re-generation, with equal probability. In the crossover operation, all sub-lists are affected. Choosing with equal probability, it will either perform a single point crossover that splits between the sub-lists of $K_{\text{3D, table}}$ or it will perform individual crossover operations within all of the sub-lists separately and combine the results to form two offspring configurations. The operations performed inside each individual sub-list during mutation and crossover are the same as described for the sparse table configuration in Section 5.3.3, but without the interpolation flag, which is not present in this case.

If the interpolation is included in the design space, the three sub-lists in $K_{\text{3D, interp}}$ are handled similarly. The mutation will randomly choose one sub-list and perform full or partial re-generation of the parameters, where the partial re-generation keeps both the first multiplier setup $M_{\text{t,h,v}}(1)$ and the width of the first set of intermediate results $F_{\text{in}}(1)$ and newly generates the remaining sub-list, namely $M_{\text{t,h,v}}(2)$, $F_{\text{in}}(2)$ and $M_{\text{t,h,v}}(3)$. The crossover operation on the other hand performs a single-point crossover to mix the sub-lists from two parents.

Lastly, in the full design space, a mutation on $K_{\text{3D, full}}$ will randomly choose to mutate either $K_{\text{3D, table}}$ or $K_{\text{3D, interp}}$ using the methods described above, while the crossover always affects both of these high-level sub-lists.

5.4.4 Optimization Setup

The extraction of internal toggle rates was performed with the two baseline configurations using 16 respectively 32 segments per dimension. It resulted in values from 0.14 to 0.27 among the two baseline setups and between the images from the ARRI set [134], with an average of 0.22. The application-specific characteristic per-unit power consumption was extracted using the two baseline setups and four additional configurations selected from 10 random ones to cover a wide range in their resource consumption.

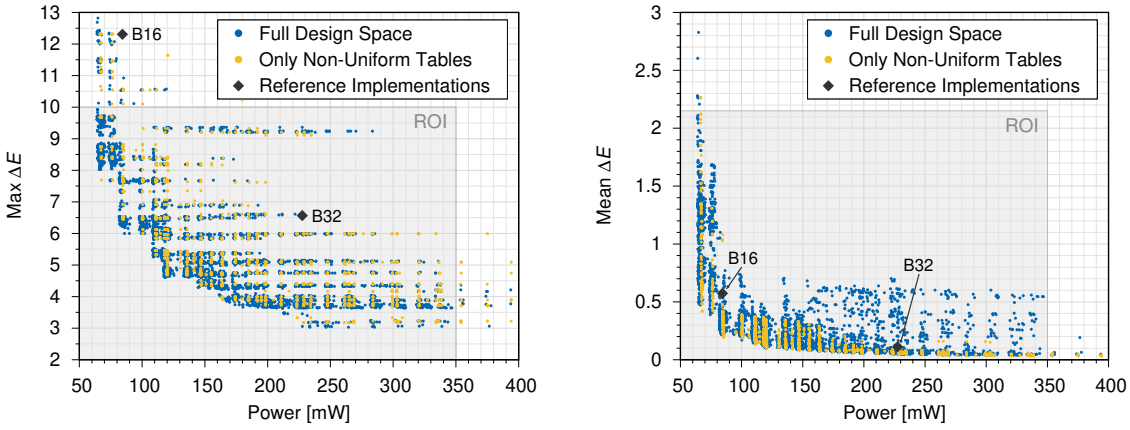
As with the previous case study, since the application output images are intended for consumption by humans, we employ the ΔE color accuracy measure [139]. However, in this case, we treat the minimization of the maximum ΔE and the mean ΔE as equivalent quality targets. Therefore, we use a combined quality training set that contains a synthetically generated subset using 128 sampling steps in each dimension and a real-world subset containing 64^3 randomly sampled inputs from the *Color Wheel* image in logC encoding to match the scene-referred input space of the application (cf. Section 3.4.3). While the worst-case quality degradation in terms of the maximum ΔE is estimated across both subsets, only the results corresponding to the real-world subset are used to estimate the mean ΔE. In contrast to the previous case studies, there is no clearly defined reference hardware implementation. Instead, we calculate the golden output with a floating-point software implementation that uses the full 3D-LUT reference data with 64 segments in each dimension and round the results to obtain colors with 12 bits per component.

The DSE targets the objectives *minimize*(power), *minimize*(max ΔE) and *minimize*(mean ΔE), with the quality targets calculated as described above, and the GA loop is iterated until 500 generations have evolved in each run. For the ROI in the objective space, we extend the maximum ΔE boundary to 10, which lies between the estimated values of the baseline solutions while keeping the mean ΔE boundary at the human perceptibility threshold of 2.15 as in the previous case study [160]. Due to the lack of a clear reference implementation, we set the ROI boundary in the power dimension to 350 mW, which exceeds the estimated power consumption of the 32-segment baseline implementation by a factor of approximately 1.5, to allow the optimization to explore solutions around the baseline implementations.

5.4.5 DSE Results

The aggregated results of 10 runs each for the two design spaces are depicted in Figure 5.17, split into two different two-dimensional projections of the solution space, namely max ΔE over power in Figure 5.17a and mean ΔE over power in Figure 5.17b.

The plots show that the solutions cover a wide range of relevant solutions within the ROI. Overall, the restricted non-uniform table design space and the full design space offer similar trade-offs across the ROI. However, a pattern can be seen in the results with the restricted design space, with noticeable gaps between different solution clusters. While this tendency is also present in the full design space results, it is less pronounced, suggesting that the additional approximations allow for more nuances in the trade-off. At most power levels, the full design space enables small improvements in terms of max ΔE errors while the restricted design space tends towards lower mean ΔE values. However, most of the solutions show mean ΔE values below 1, which is sufficient in most situations. The fact that both design spaces cover roughly the same area supports the notion that the quality-resource trade-off offered by the 3D-LUT application is largely dominated by the table segmentation which controls the memory consumption.



(a) Projection of max ΔE over power

(b) Projection of mean ΔE over power

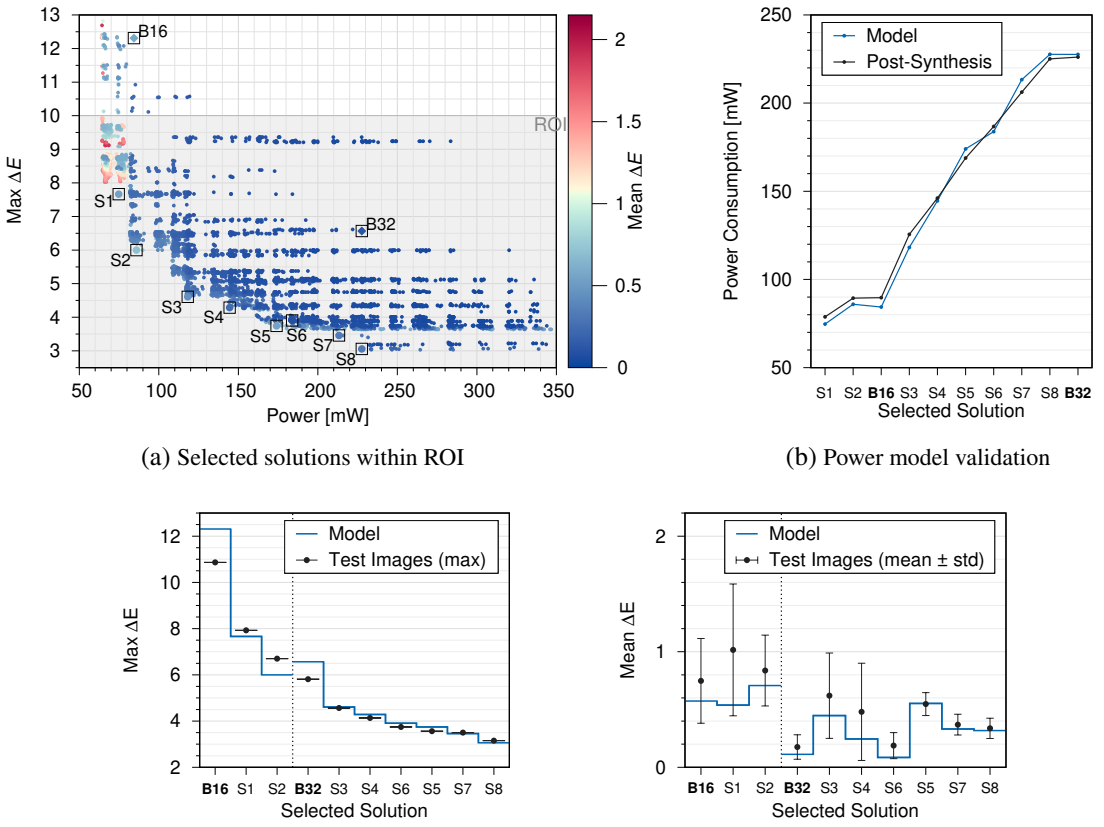
Figure 5.17: DSE results for CS3 obtained for the compared design spaces, i.e. with all approximations enabled (blue) or using only non-uniform table segmentations (yellow), together with the estimated values for the baseline solutions with 16 respectively 32 uniform segments (black)

The baseline solutions with 16 respectively 32 uniform segments in each direction are marked as *B16* and *B32* in the plots of Figure 5.17. When comparing the results to the performance of the baseline solutions, the plots suggest that at a similar or lower power consumption, the worst case quality loss can be reduced significantly while keeping the average quality at a similar or slightly worse level. Also, both design spaces offer considerably finer gradation in tuning the trade-off compared to the baseline solutions which do not offer intermediate configurations.

5.4.6 Model Validation

Similar to the previous case studies, we have selected 8 solutions from the results obtained with the full design space for further investigation. Both the selected solutions and the baseline implementations are highlighted in Figure 5.18, which is color coded to show all three objectives at once. As the plot shows, solutions S1 and S2 directly compete with the lower quality baseline implementation B16, while the other solutions S3 through S8 compete with the higher quality baseline implementation B32. Detailed validation data of these solutions in terms of area and power consumption, maximum speed and application quality are listed in Table 5.7.

Area and Power Consumption The estimation of consumed logic resources shows similarly high accuracy as for the previous case studies, as shown in Table 5.7. For LUTs, the average relative error among all solutions is 0.98% while in the worst case, the deviation is 2.13% (at S2). On the other hand, registers are estimated with an average deviation of 0.39% with a worst-case error of 0.77% (also at S2). The table shows that while the memory consumption differs widely between the different solutions, only a limited number of DSPs is replaced by approximate multipliers. This coincides with the observation made above that the design space is dominated by the table segmentation, while the remaining approximations may be used to achieve fine further improvements.



(a) Selected solutions within ROI

(b) Power model validation

(c) Quality model validation (max ΔE)

(d) Quality model validation (mean ΔE)

Figure 5.18: CS3: Selected solutions for model validation from ROI (a), comparison with post-synthesis power values (b) and quality validation with test images from the validation set (c/d)

Table 5.7: Area, power and quality data of the selected points for CS3

Sel. Point Index	FPGA Resources						Power		Speed	Quality			
	DSPs[1]	BRAMs[1] [Units/Bits]	LUTs		Registers		[mW]		[MHz]	max ΔE		mean ΔE	
			Model	Synth	Model	Synth	Model	Synth	Synth	Train	Test	Train	Test
S1	21	12 / 151 200	1055	1040	1697	1688	74.74	78.79	281.18	7.66	7.92	0.54	1.05
S2	16	16 / 275 400	1390	1361	1831	1817	85.96	89.41	283.49	6.00	5.64	0.71	0.85
S3	20	32 / 527 544	1113	1101	1688	1679	118.14	125.54	259.23	4.61	4.19	0.45	0.64
S4	21	40 / 692 496	1435	1406	2028	2020	144.55	146.15	284.46	4.29	4.12	0.24	0.50
S5	18	52 / 910 800	1770	1755	2142	2138	174.02	168.86	275.14	3.74	3.48	0.55	0.55
S6	21	56 / 990 000	1772	1775	2099	2091	183.81	186.75	281.47	3.91	3.75	0.09	0.19
S7	16	68 / 1 200 600	2145	2162	2263	2257	213.28	206.24	279.84	3.46	3.50	0.33	0.37
S8	19	76 / 1 330 056	1849	1839	2144	2130	227.65	225.05	283.44	3.06	2.36	0.32	0.34
B16	21	16 / 186 048	1071	1067	1745	1743	84.35	89.69	284.18	12.31	10.87	0.57	0.75
B32	21	76 / 1 330 560	1770	1764	2117	2116	227.60	226.11	285.24	6.57	5.81	0.11	0.18

[1] The estimation of consumed DSPs and BRAMs by the proposed model matches the post-synthesis results for all systems

The power trend across all solutions, as captured by the model and post-synthesis, is additionally visualized in Figure 5.18b. The overall trend and the relations between individual solutions are mostly captured correctly by the model, except for the relations between S2/B16 as well as S8/B32, which are slightly reversed in the post-synthesis results compared to the model. However, both of these pairs have very similar absolute power values. Overall, the average and worst-case estimation errors are reasonably low at 3.27% respectively 6.33%.

Speed In contrast to the previous case studies, there is no clear relation between the position of a solution on the quality-resource trade-off curve and the achievable speed. Most solutions are able to operate at speeds around 280 MHz. However, there is one outlier, S3, which only reaches 259.23 MHz and therefore fails to meet the target frequency of 266.66 MHz. This shows that the employed mechanism to ensure meeting the timing requirements, while working adequately in most cases, is not perfect. In such cases, the designer can try additional seeds[3], increase the fitting effort of the tool, implement manual timing optimizations or choose another solution.

Quality In this case study, we analyze both the worst-case quality degradation in terms of maximum ΔE and the average quality loss in terms of mean ΔE. Figures 5.18c and 5.18d visualize the respective comparisons between model and validation data. As mentioned above, we can split the selected solutions into two groups, with S1-S2 competing against B16 and S3-S8 competing against B32. To be able to analyze the comparison within these groups more easily, the solutions are sorted accordingly in the figures and the groups are visually separated by a vertical line. Note that solutions S5 and S6 are reversed in the plots to match their trend in the maximum ΔE objective.

It can be seen in Figure 5.18c that the model predicts higher worst-case errors as seen in the test images for the baseline solutions, while it underestimates them for some of the selected solutions, especially S1 and S2. However, we can observe that the selected solutions enable a significant reduction of the maximum ΔE compared to the respective baseline implementation in both groups.

Regarding the average quality loss, the conclusion is less clear. As a general trend, if the variation between individual test images is high, the model predictions tend to be less accurate, which is similar to what could be observed in the quality validation of CS1 (cf. Section 5.2.5). However, the estimation is always within the standard deviation across the test images. In both comparison groups we can see that the reduction in maximum ΔE is afforded by some degradation in mean ΔE. However, the mean ΔE is generally at a rather low level across all solutions with reported averages staying below 1 in most cases.

Solution S6 should be highlighted as it achieves nearly the same average quality as B32 while reducing the maximum ΔE in the validation set from 5.81 to 3.75 and simultaneously reducing the post-synthesis power consumption by 17.41%.

5.5 Summary

This chapter presented three case studies that target different real-world applications for processing image colors in digital camera systems. The proposed framework was used to integrate approximations of different types, namely precision scaling, approximate arithmetics and hierarchically

[3]The highest speed reported for S3 across the 10 seeds used to aggregate the post-synthesis data is 265.04 MHz while the data reported in Table 5.7 represents the average across all seeds

segmented sparse tables into those applications, and to optimize their parameterization globally across the respective DFGs.

The experimental results show that the proposed framework is able to effectively generate well populated Pareto-fronts across given ROIs in the objective space. Comparing the full design space with limited sub-spaces that cover only single approximation types showcases the benefits of combining approximations across multiple types in the same application to unlock larger resource savings. The proposed framework allows such combinations by using flexibly sizable approximate components and by considering the effects of approximations on internal signal widths and the related implications for connected system components.

For each case study, selected solutions were synthesized for a specific FPGA device to validate the resource consumption. The reported data shows that the models are highly accurate, with worst-case deviations in area and power consumption rarely exceeding a relative error of 5%. Regarding the target operating frequency, all selected solutions except one were able to meet or exceed the timing requirements, indicating that the proposed mechanism for achieving a specific minimum speed generally works well. Furthermore, a comparison of the quality metric values estimated during the DSE from training sets of restricted size with validation data obtained for 12 full-size images showed that while the quality prediction is not perfect, it provides a good indication of what can be expected in real images in terms of quality.

In practical scenarios, we recommend using the proposed framework to prepare the design space and run the DSE to obtain general information about the possible quality-resource trade-offs offered by the target application. Based upon the model data, the designer is able to narrow down a small set of suitable candidates close to the desired trade-off point and make a final decision based on actual synthesis results.

Chapter 6

Conclusion & Outlook

Current research on approximate computing opens new possibilities for improving the efficiency of computational systems, promising solutions for designers that face growing demands for their applications while technological improvements are diminishing. The aim of the framework proposed in this dissertation is to bridge the gap between the large research body containing proposals of individual approximation methods across different categories and their practical, efficient and symbiotic integration into real-world applications implemented in FPGA devices. This chapter summarizes the work presented in this dissertation together with the main observed experimental findings before outlining topics that remain open for future work.

6.1 Thesis Summary

This dissertation proposes a framework for systematic integration and parameterization of approximation methods into FPGA-based applications, which can be divided into three main design phases, namely the implementation of scalable and reusable approximate components, their integration into target applications and lastly their optimal parameterization within the design space spanned across the approximated application.

First, Chapter 2 laid the foundation of the framework by forming a library of approximate components that can be flexibly integrated into FPGA-based applications. It provided an extensive survey and categorization of approximation methods proposed in related literature which target hardware implementations. These methods were filtered according to their suitability for implementation in typical FPGA architectures and their relevance for the target application scope, focusing on in-camera pixel stream processing pipelines. Selected promising approximation techniques from the categories of *approximate arithmetic units* and *table-based methods* were adapted for the use in FPGA-based systems and implemented in a flexibly scalable manner so that they can be used in conjunction with *fine-grain precision scaling*. The characteristics of these components, representing their isolated quality-resource trade-off performance, were extracted using an automatized workflow. Based on the characterization data, competing versions of adders and multipliers were compared extensively, which provided valuable insights regarding their ability to translate to FPGA architectures. Furthermore, ML-based models were trained using the characterization data to enable a fast estimation of resource usage across various component sizes and approximation strengths. Finally, the parameterizable hardware implementations, resource models, and behavioral software models of the selected components were integrated into a library that provides systematic interfaces for the integration into various target applications, completing the first phase of the proposed design flow.

Next, Chapter 3 proposed a DFG-based representation of target applications, which enables the integration of approximations and setting their parameter ranges via user annotations. Hence, an *annotated DFG* models the entire design space of an approximated application, and a *candidate DFG* represents a single specific parameterization across all components. Furthermore, application-level models for the resource consumption and quality degradation associated with any candidate DFG were proposed. In terms of application resources, the area consumption is estimated first in a divide-and-conquer fashion from the individual components in the candidate DFG, and the power consumption is derived from the estimated area using the characteristic per-unit power consumption of the different FPGA resource types. Covering the other side of the approximate computing trade-off, a simulation-based quality model was proposed that allows the designer to integrate their preferred choice of any reference quality metric into the fitness estimation. Additionally, relevant implications for choosing suitable training input data for the quality model were discussed, and, focusing on the applications targeted and studied in this dissertation, two types of data sets, namely *synthetic* and *real-world* inputs, were proposed and their properties analyzed.

Chapter 4 presented the DSE procedure in the final phase of the design flow. After reviewing related approaches a GA-based approach was selected, which is based on related work that was developed in a collaborative project [5]. An overview of the selected method was given and its integration into the framework described.

To demonstrate the proposed framework, Chapter 5 presented three case studies targeting different real-world color processing applications. Experimental results show that the framework is able to provide a densely populated front of relevant solutions, covering a wide range of different quality-resource trade-offs to choose from. Additionally, the DSE was executed across differently annotated DFGs, each limiting the design space to just one singular type of approximation. The results of those experiments have shown that a symbiotic combination of different approximation types can achieve larger benefits than using only a single type, which is a common limitation in related work. Lastly, the accuracy of the proposed models was analyzed by selecting a range of relevant solutions from the DSE results and validating the modeled fitness values against post-synthesis data and the actual quality observed in a set of real images. The comparison showed that the fitness estimation with the proposed models holds up well to the validation data and therefore is suitable for guiding the DSE and for providing a good indication of the expected values in final implementations.

6.2 Open Topics and Future Work

While the case studies have shown the functionality of the proposed framework and demonstrated its proficiency for integrating approximations across different types into FPGA-based applications, there is still room for improvement. On the one hand, the efficiency of the framework may be enhanced further. On the other hand, its functionality can be extended to support a wider range of applications and design workflows as well as directly targeting improvements in computational speed. Potential directions for future work on these topics are outlined in the paragraphs below.

Potential efficiency improvements The comparison of the full design space and the restricted design spaces have shown that some solutions with low resource benefits but high application quality were only found in the restricted design spaces. Ideally, the DSE results from the full design space, which theoretically contains these solutions as well, should cover all Pareto-Optimal

solutions. There is still unexploited potential in tuning the optimization methodology. First, the GA hyperparameters such as the mutation/crossover probabilities and the number of selected parents and created offspring per generation were set to standard values, but could be improved by performing an offline hyper-parameter search or by employing a dynamic adaption scheme during the DSE [153]. Furthermore, the parameterization order of the DFG configuration (see Section 3.2.1) and the derived genetic operations (see Section 4.2.2) inherently prevent the modification of parameters early in the propagation chain without affecting subsequent ones. To overcome this limitation, the mutation operators could be refined to include the possibility of changing early parameters. However, this must be done carefully, as the modification of an early parameter may change the meaning or interpretation of the later parameters, which may need to be adapted accordingly to preserve their original behavior.

Supporting Spatial Processing The application scope of this dissertation focuses on FPGA-based image processing systems which employ pixel streaming pipelines. As described in Section 3.1, the proposed framework is currently limited to operations on individual pixels, as used in color transformations. However, some applications, e.g. FIR filters or noise reduction, need the information of multiple neighboring pixels within a given window [163]. Typically, such a system is represented in related work by treating each neighboring pixel as separate external system input [124, 151, 152]. To achieve similar functionality, the DFG-based behavioral simulation could easily be extended to include an external input management that feeds correctly shifted pixel streams to multiple inputs. However, in actual hardware implementations of spatially operating stream processing systems, the pixel stream must be cached in local memory, e.g. BRAMs [163]. Typically, to open a window of size $W \times W$, $W-1$ entire rows or columns must be buffered, depending on the streaming order, consuming a considerable amount of memory resources. For images with large resolutions this may even dominate the overall resource consumption of the entire system. Such considerations are completely missing from related literature. To overcome this limitation, additional component models for row/column buffers could be added to the component library which would allow to truthfully capture all implications introduced by spatial processing.

Integration into HLS Workflows In the current state of the framework, the DFG representing the target application has to be written manually by the designer using the interfaces provided by the framework. In contrast, many related works already cover the integration of their design methodologies into established HLS workflows [150, 152]. Similar extensions could be made for the framework to improve its ease of use. Necessary modifications to achieve such an integration comprise the automatic construction of the DFG from a high-level description or an intermediate representation from the respective HLS workflow, if available, and a user interface to facilitate the annotation of DFG components. The latter task could also be solved by using software-level annotations in the high-level code that are automatically translated to DFG annotations.

Including Computational Speed as Full Optimization Target Lastly, the proposed framework only aims to ensure the satisfaction of a given minimum timing constraint by modeling the delay of the largest combinatorial blocks, which were the multipliers in the studied applications. However, computational speed is typically part of the traditional hardware design trade-off (cf. Figure 1.1) and is often also included in the approximate computing trade-off space [18]. The model validation results of the first two case studies reported in Sections 5.2.5 and 5.3.6 indicate

that performance benefits due to the use of approximations are possible in FPGA-based pipelined stream processing systems. However, in order to integrate speed as a full target objective, a more sophisticated delay model that covers all paths of the application needs to be implemented. To that end, the approach used in AxHLS could be adopted, which estimates the delay in a divide-and-conquer manner by summing up individual component delays along the critical path [152].

Appendix A

ARRI Image Set

(a) Akademie (b) Arri (c) Church

(d) Color Test Chart (e) Face (f) Lake Locked

(g) Lake Pan (h) Night at Odeonsplatz (i) Swimming Pool

(j) Sharpness Chart (k) Night at Siegestor (l) Color Wheel

Figure A.1: Images from the ARRI Test Set in Rec709 display-referred format [134]

Bibliography

Publications by the author

[1] S. Conrady, M. Manuel, A. Kreddig, and W. Stechele, "LCS-based automatic configuration of approximate computing parameters for FPGA system designs," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '19)*, Association for Computing Machinery, Jul. 13, 2019, pp. 1271–1279. DOI: 10.1145/3319619.3326820.

[2] M. Manuel, A. Kreddig, S. Conrady, N. Anh Vu Doan, and W. Stechele, "Model-Based Design Space Exploration for Approximate Image Processing on FPGA," in *2020 IEEE Nordic Circuits and Systems Conference (NorCAS)*, Oct. 2020, pp. 1–7. DOI: 10.1109/NorCAS51424.2020.9265138.

[3] N. A. V. Doan, M. Manuel, S. Conrady, A. Kreddig, and W. Stechele, "Parameter Optimization of Approximate Image Processing Algorithms in FPGAs," in *2020 Eighth International Symposium on Computing and Networking Workshops (CANDARW)*, Nov. 2020, pp. 74–80. DOI: 10.1109/CANDARW51189.2020.00026.

[4] S. Conrady, A. Kreddig, M. Manuel, N. A. V. Doan, and W. Stechele, "Model-Based Design Space Exploration for FPGA-based Image Processing Applications Employing Parameterizable Approximations," *Microprocessors and Microsystems*, vol. 87, p. 104 386, Nov. 2021. DOI: 10.1016/j.micpro.2021.104386.

[5] M. Manuel, A. Kreddig, S. Conrady, N. A. V. Doan, and W. Stechele, "Region of Interest-Based Parameter Optimization for Approximate Image Processing on FPGAs," *International Journal of Networking and Computing*, vol. 11, no. 2, pp. 438–462, 2021. DOI: 10.15803/ijnc.11.2_438.

[6] A. Kreddig, S. Conrady, M. Manuel, and W. Stechele, "A Framework for Hardware-Accelerated Design Space Exploration for Approximate Computing on FPGA," in *2021 24th Euromicro Conference on Digital System Design (DSD)*, Sep. 2021, pp. 1–8. DOI: 10.1109/DSD53832.2021.00010.

General Publications

[7] M. Bohr, "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, 2007. DOI: 10.1109/N-SSC.2007.4785534.

[8] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11, New York, NY, USA: Association for Computing Machinery, Jun. 2011, pp. 365–376. DOI: 10.1145/2000064.2000108.

[9] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing and the quest for computing efficiency," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun. 2015. DOI: 10.1145/2744769.2751163.

[10] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate Computing: A Survey," *IEEE Design Test*, vol. 33, no. 1, pp. 8–22, Feb. 2016. DOI: 10.1109/MDAT.2015.2505723.

[11] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, May 2013, pp. 1–9. DOI: 10.1145/2463209.2488873.

[12] EMVA 1288 Working Group, "EMVA Standard 1288: Standard for Characterization of Image Sensors and Cameras, Release 4.0 General," European Machine Vision Association, Standard, Jun. 2021. [Online]. Available: https://www.emva.org/standards-technology/emva-1288/ (visited on 04/08/2022).

[13] T. Seybold, "Digital Motion Picture Camera Denoising," Ph.D. dissertation, Technische Universität München, München, 2015. [Online]. Available: http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20151215-1273923-1-5 (visited on 04/08/2022).

[14] D. G. Bailey, "Image Processing," in *Design for Embedded Image Processing on FPGAs*, IEEE, 2011, pp. 1–19. DOI: 10.1002/9780470828519.ch1.

[15] Intel Corporation, *Intel Arria 10 FPGAs*. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/fpga/arria/10.html (visited on 04/08/2022).

[16] Intel Corporation, *Intel Arria 10 SoC Development Kit*. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/arria/10-sx.html (visited on 04/08/2022).

[17] I. Akturk, K. Khatamifard, and U. R. Karpuzcu, "On Quantification of Accuracy Loss in Approximate Computing," in *12th Annual Workshop on Duplicating, Deconstructing and Debunking (WDDD)*, Jun. 2015. [Online]. Available: http://altai.ece.umn.edu/accurax/accurax.html (visited on 04/08/2022).

[18] S. Mittal, "A Survey of Techniques for Approximate Computing," *ACM Comput. Surv.*, vol. 48, no. 4, 62:1–62:33, Mar. 2016. DOI: 10.1145/2893356.

[19] H. B. Barua and K. C. Mondal, "Approximate Computing: A Survey of Recent Trends - Bringing Greenness to Computing and Communication," *Journal of The Institution of Engineers (India): Series B*, vol. 100, no. 6, pp. 619–626, Dec. 2019. DOI: 10.1007/s40031-019-00418-8.

[20] G. Rodrigues, F. Lima Kastensmidt, and A. Bosio, "Survey on Approximate Computing and Its Intrinsic Fault Tolerance," *Electronics*, vol. 9, no. 4, p. 557, Apr. 2020. DOI: 10.3390/electronics9040557.

[21] A. B. Kahng, S. Kang, R. Kumar, and J. Sartori, "Slack redistribution for graceful degradation under voltage overscaling," in *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2010, pp. 825–831. DOI: 10.1109/ASPDAC.2010.5419690.

[22] D. Mohapatra, V. K. Chippa, A. Raghunathan, and K. Roy, "Design of voltage-scalable meta-functions for approximate computing," in *2011 Design, Automation Test in Europe*, Mar. 2011, pp. 1–6. DOI: 10.1109/DATE.2011.5763154.

[23] G. Zervakis, F. Ntouskas, S. Xydis, D. Soudris, and K. Pekmestzi, "VOSsim: A Framework for Enabling Fast Voltage Overscaling Simulation for Approximate Computing Circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. PP, no. 99, pp. 1–5, 2018. DOI: 10.1109/TVLSI.2018.2803202.

[24] G. Zervakis, S. Xydis, D. Soudris, and K. Pekmestzi, "Multi-Level Approximate Accelerator Synthesis Under Voltage Island Constraints," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 66, no. 4, pp. 607–611, Apr. 2019. DOI: 10.1109/TCSII.2018.2869025.

[25] D. U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides, "Accuracy-Guaranteed Bit-Width Optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1990–2000, Oct. 2006. DOI: 10.1109/TCAD.2006.873887.

[26] W. G. Osborne, J. G. F. Coutinho, R. C. C. Cheung, W. Luk, and O. Mencer, "Instrumented Multi-Stage Word-Length Optimization," in *2007 International Conference on Field-Programmable Technology*, Dec. 2007, pp. 89–96. DOI: 10.1109/FPT.2007.4439236.

[27] J. Park, J. H. Choi, and K. Roy, "Dynamic Bit-Width Adaptation in DCT: An Approach to Trade Off Image Quality and Computation Energy," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 5, pp. 787–793, May 2010. DOI: 10.1109/TVLSI.2009.2016839.

[28] S. Lee and A. Gerstlauer, "Fine Grain Precision Scaling for Datapath Approximations in Digital Signal Processing Systems," in *VLSI-SoC: At the Crossroads of Emerging Trends*, A. Orailoglu, H. F. Ugurdag, L. M. Silveira, M. Margala, and R. Reis, Eds., ser. IFIP Advances in Information and Communication Technology, Cham: Springer International Publishing, 2015, pp. 119–143. DOI: 10.1007/978-3-319-23799-2_6.

[29] A. Lingamneni, C. Enz, J. L. Nagel, K. Palem, and C. Piguet, "Energy parsimonious circuit design through probabilistic pruning," in *2011 Design, Automation Test in Europe*, Mar. 2011, pp. 1–6. DOI: 10.1109/DATE.2011.5763130.

[30] J. Schlachter, V. Camus, K. V. Palem, and C. Enz, "Design and Applications of Approximate Circuits by Gate-Level Pruning," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 5, pp. 1694–1702, May 2017. DOI: 10.1109/TVLSI.2017.2657799.

[31] D. May and W. Stechele, "Design of fine-grained sequential approximate circuits using probability-aware fault emulation," in *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, Jul. 2015, pp. 73–78. DOI: 10.1109/ISLPED.2015.7273493.

[32] Z. Zhang, Y. He, J. He, X. Yi, Q. Li, and B. Zhang, "Optimal Slope Ranking: An Approximate Computing Approach for Circuit Pruning," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2018, pp. 1–4. DOI: 10.1109/ISCAS.2018.8351238.

[33] I. Scarabottolo, G. Ansaloni, and L. Pozzi, "Circuit carving: A methodology for the design of approximate hardware," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2018, pp. 545–550. DOI: 10.23919/DATE.2018.8342067.

[34] D. Shin and S. K. Gupta, "Approximate logic synthesis for error tolerant applications," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, Mar. 2010, pp. 957–960. DOI: 10.1109/DATE.2010.5456913.

[35] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "SALSA: Systematic logic synthesis of approximate circuits," in *DAC Design Automation Conference 2012*, Jun. 2012, pp. 796–801. DOI: 10.1145/2228360.2228504.

[36] J. Miao, A. Gerstlauer, and M. Orshansky, "Multi-level approximate logic synthesis under general error constraints," in *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2014, pp. 504–510. DOI: 10.1109/ICCAD.2014.7001398.

[37] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2013, pp. 1367–1372. DOI: 10.7873/DATE.2013.280.

[38] L. Sekanina and Z. Vasicek, "Approximate circuit design by means of evolvable hardware," in *2013 IEEE International Conference on Evolvable Systems (ICES)*, Apr. 2013, pp. 21–28. DOI: 10.1109/ICES.2013.6613278.

[39] Z. Vasicek and L. Sekanina, "Evolutionary Approach to Approximate Digital Circuits Design," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 3, pp. 432–444, Jun. 2015. DOI: 10.1109/TEVC.2014.2336175.

[40] L. Sekanina, Z. Vasicek, and V. Mrazek, "Automated Search-Based Functional Approximation for Digital Circuits," in *Approximate Circuits: Methodologies and CAD*, S. Reda and M. Shafique, Eds., Cham: Springer International Publishing, 2019, pp. 175–203. DOI: 10.1007/978-3-319-99322-5_9.

[41] Z. Vasicek and L. Sekanina, "Search-based synthesis of approximate circuits implemented into FPGAs," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug. 2016, pp. 1–4. DOI: 10.1109/FPL.2016.7577305.

[42] S. Hashemi, H. Tann, and S. Reda, "BLASYS: Approximate Logic Synthesis Using Boolean Matrix Factorization," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, Jun. 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465702.

[43] 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)*, Mar. 2014, pp. 1–6. DOI: 10.7873/DATE.2014.377.

[44] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-Power Digital Signal Processing Using Approximate Adders," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 1, pp. 124–137, Jan. 2013. DOI: 10.1109/TCAD.2012.2217962.

[45] Z. Yang, A. Jain, J. Liang, J. Han, and F. Lombardi, "Approximate XOR/XNOR-based adders for inexact computing," in *2013 13th IEEE International Conference on Nanotechnology (IEEE-NANO 2013)*, Aug. 2013, pp. 690–693. DOI: 10.1109/NANO.2013.6720793.

[46] H. A. F. Almurib, T. N. Kumar, and F. Lombardi, "Inexact designs for approximate low power addition by cell replacement," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2016, pp. 660–665. [Online]. Available: https://dl.acm.org/doi/10.5555/2971808.2971962 (visited on 04/08/2022).

[47] B. S. Prabakaran *et al.*, "DeMAS: An efficient design methodology for building approximate adders for FPGA-based systems," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2018, pp. 917–920. DOI: 10.23919/DATE.2018.8342140.

[48] N. Zhu, W. L. Goh, W. Zhang, K. S. Yeo, and Z. H. Kong, "Design of Low-Power High-Speed Truncation-Error-Tolerant Adder and Its Application in Digital Signal Processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 8, pp. 1225–1229, Aug. 2010. DOI: 10.1109/TVLSI.2009.2020591.

[49] D. Celia, V. Vasudevan, and N. Chandrachoodan, "Optimizing power-accuracy trade-off in approximate adders," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2018, pp. 1488–1491. DOI: 10.23919/DATE.2018.8342248.

[50] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas, "Bio-Inspired Imprecise Computational Blocks for Efficient VLSI Implementation of Soft-Computing Applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 57, no. 4, pp. 850–862, Apr. 2010. DOI: 10.1109/TCSI.2009.2027626.

[51] P. Albicocco, G. C. Cardarilli, A. Nannarelli, M. Petricca, and M. Re, "Imprecise arithmetic for low power image processing," in *2012 Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, Nov. 2012, pp. 983–987. DOI: 10.1109/ACSSC.2012.6489164.

[52] A. Dalloo, A. Najafi, and A. Garcia-Ortiz, "Systematic Design of an Approximate Adder: The Optimized Lower Part Constant-OR Adder," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 8, pp. 1595–1599, Aug. 2018. DOI: 10.1109/TVLSI.2018.2822278.

[53] P. Balasubramanian, R. Nayar, D. L. Maskell, and N. E. Mastorakis, "An Approximate Adder With a Near-Normal Error Distribution: Design, Error Analysis and Practical Application," *IEEE Access*, vol. 9, pp. 4518–4530, 2021. DOI: 10.1109/ACCESS.2020.3047651.

[54] J. Echavarria, S. Wildermann, A. Becher, J. Teich, and D. Ziener, "FAU: Fast and error-optimized approximate adder units on LUT-Based FPGAs," in *2016 International Conference on Field-Programmable Technology (FPT)*, Dec. 2016, pp. 213–216. DOI: 10.1109/FPT.2016.7929536.

[55] N. Zhu, W. L. Goh, and K. S. Yeo, "An enhanced low-power high-speed Adder For Error-Tolerant application," in *Proceedings of the 2009 12th International Symposium on Integrated Circuits*, Dec. 2009, pp. 69–72. [Online]. Available: https://ieeexplore.ieee.org/document/5403865 (visited on 04/08/2022).

[56] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *DAC Design Automation Conference 2012*, Jun. 2012, pp. 820–825. DOI: 10.1145/2228360.2228509.

[57] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu, "On reconfiguration-oriented approximate adder design and its application," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2013, pp. 48–54. DOI: 10.1109/ICCAD.2013.6691096.

[58] M. Shafique, W. Ahmad, R. Hafiz, and J. Henkel, "A Low Latency Generic Accuracy Configurable Adder," in *Proceedings of the 52Nd Annual Design Automation Conference*, ser. DAC '15, New York, NY, USA: ACM, 2015, 86:1–86:6. DOI: 10.1145/2744769.2744778.

[59] 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*, Mar. 2017, pp. 258–261. DOI: 10.23919/DATE.2017.7926993.

[60] 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*, vol. 10, no. 4, pp. 406–418, Dec. 2020. DOI: 10.1109/JETCAS.2020.3032495.

[61] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading Accuracy for Power with an Underdesigned Multiplier Architecture," in *2011 24th Internatioal Conference on VLSI Design*, Jan. 2011, pp. 346–351. DOI: 10.1109/VLSID.2011.51.

[62] S. Rehman, W. El-Harouni, M. Shafique, A. Kumar, J. Henkel, and J. Henkel, "Architectural-space exploration of approximate multipliers," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2016, pp. 1–8. DOI: 10.1145/2966986.2967005.

[63] S. Ullah *et al.*, "Area-optimized low-latency approximate multipliers for FPGA-based hardware accelerators," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18, New York, NY, USA: Association for Computing Machinery, Jun. 2018, pp. 1–6. DOI: 10.1145/3195970.3195996.

[64] F. Farshchi, M. S. Abrishami, and S. M. Fakhraie, "New approximate multiplier for low power digital signal processing," in *The 17th CSI International Symposium on Computer Architecture Digital Systems (CADS 2013)*, Oct. 2013, pp. 25–30. DOI: 10.1109/CADS.2013.6714233.

[65] K. Y. Kyaw, W. L. Goh, and K. S. Yeo, "Low-power high-speed multiplier for error-tolerant application," in *2010 IEEE International Conference of Electron Devices and Solid-State Circuits (EDSSC)*, Dec. 2010, pp. 1–4. DOI: 10.1109/EDSSC.2010.5713751.

[66] S. Narayanamoorthy, H. A. Moghaddam, Z. Liu, T. Park, and N. S. Kim, "Energy-Efficient Approximate Multiplication for Digital Signal Processing and Classification Applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 6, pp. 1180–1184, Jun. 2015. DOI: 10.1109/TVLSI.2014.2333366.

[67] S. Hashemi, R. I. Bahar, and S. Reda, "DRUM: A Dynamic Range Unbiased Multiplier for approximate applications," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2015, pp. 418–425. DOI: 10.1109/ICCAD.2015.7372600.

[68] R. Zendegani, M. Kamal, M. Bahadori, A. Afzali-Kusha, and M. Pedram, "RoBA Multiplier: A Rounding-Based Approximate Multiplier for High-Speed yet Energy-Efficient Digital Signal Processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 2, pp. 393–401, Feb. 2017. DOI: 10.1109/TVLSI.2016.2587696.

[69] B. Garg and S. Patel, "Reconfigurable Rounding Based Approximate Multiplier for Energy Efficient Multimedia Applications," *Wireless Personal Communications*, vol. 118, no. 2, pp. 919–931, May 2021. DOI: 10.1007/s11277-020-08051-1.

[70] J. N. Mitchell, "Computer Multiplication and Division Using Binary Logarithms," *IRE Transactions on Electronic Computers*, vol. EC-11, no. 4, pp. 512–517, Aug. 1962. DOI: 10.1109/TEC.1962.5219391.

[71] M. S. Kim, A. A. D. Barrio, L. T. Oliveira, R. Hermida, and N. Bagherzadeh, "Efficient Mitchell's Approximate Log Multipliers for Convolutional Neural Networks," *IEEE Transactions on Computers*, vol. 68, no. 5, pp. 660–675, May 2019. DOI: 10.1109/TC.2018.2880742.

[72] H. Saadat, H. Bokhari, and S. Parameswaran, "Minimally Biased Multipliers for Approximate Integer and Floating-Point Multiplication," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2623–2635, Nov. 2018. DOI: 10.1109/TCAD.2018.2857262.

[73] H. Saadat, H. Javaid, A. Ignjatovic, and S. Parameswaran, "REALM: Reduced-Error Approximate Log-based Integer Multiplier," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2020, pp. 1366–1371. DOI: 10.23919/DATE48585.2020.9116315.

[74] W. Liu, J. Xu, D. Wang, C. Wang, P. Montuschi, and F. Lombardi, "Design and Evaluation of Approximate Logarithmic Multipliers for Low Power Error-Tolerant Applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 9, pp. 2856–2868, Sep. 2018. DOI: 10.1109/TCSI.2018.2792902.

[75] M. S. Ansari, B. F. Cockburn, and J. Han, "An Improved Logarithmic Multiplier for Energy-Efficient Neural Computing," *IEEE Transactions on Computers*, vol. 70, no. 4, pp. 614–625, Apr. 2021. DOI: 10.1109/TC.2020.2992113.

[76] V. Mrazek, Z. Vasicek, L. Sekanina, H. Jiang, and J. Han, "Scalable Construction of Approximate Multipliers With Formally Guaranteed Worst Case Error," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 11, pp. 2572–2576, Nov. 2018. DOI: 10.1109/TVLSI.2018.2856362.

[77] N. Brisebarre, J.-M. Muller, and A. Tisserand, "Sparse-coefficient polynomial approximations for hardware implementations," in *Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems and Computers*, vol. 1, Nov. 2004, pp. 532–535. DOI: 10.1109/ACSSC.2004.1399189.

[78] S. Chevillard, M. Joldeş, and C. Lauter, "Sollya: An Environment for the Development of Numerical Codes," in *Mathematical Software – ICMS 2010*, K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2010, pp. 28–31. DOI: 10.1007/978-3-642-15582-6_5.

[79] F. de Dinechin, M. Joldes, and B. Pasca, "Automatic generation of polynomial-based hardware architectures for function evaluation," in *ASAP 2010 - 21st IEEE International Conference on Application-Specific Systems, Architectures and Processors*, Jul. 2010, pp. 216–222. DOI: 10.1109/ASAP.2010.5540952.

[80] J. E. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Transactions on Electronic Computers*, vol. EC-8, no. 3, pp. 330–334, Sep. 1959. DOI: 10.1109/TEC.1959.5222693.

[81] J. S. Walther, "A unified algorithm for elementary functions," in *Proceedings of the May 18-20, 1971, Spring Joint Computer Conference*, ser. AFIPS '71 (Spring), New York, NY, USA: Association for Computing Machinery, May 1971, pp. 379–385. DOI: 10.1145/1478786.1478840.

[82] P. A. Kumar, "FPGA Implementation of the Trigonometric Functions Using the CORDIC Algorithm," in *2019 5th International Conference on Advanced Computing Communication Systems (ICACCS)*, Mar. 2019, pp. 894–900. DOI: 10.1109/ICACCS.2019.8728315.

[83] D. Das Sarma and D. Matula, "Faithful bipartite ROM reciprocal tables," in *Proceedings of the 12th Symposium on Computer Arithmetic*, Jul. 1995, pp. 17–28. DOI: 10.1109/ARITH.1995.465381.

[84] M. Schulte and J. Stine, "Approximating elementary functions with symmetric bipartite tables," *IEEE Transactions on Computers*, vol. 48, no. 8, pp. 842–847, Aug. 1999. DOI: 10.1109/12.795125.

[85] J. E. Stine and M. J. Schulte, "The Symmetric Table Addition Method for Accurate Function Approximation," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 21, no. 2, pp. 167–177, Jun. 1999. DOI: 10.1023/A:1008004523235.

[86] F. de Dinechin and A. Tisserand, "Multipartite table methods," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 319–330, Mar. 2005. DOI: 10.1109/TC.2005.54.

[87] D. U. Lee, R. C. C. Cheung, W. Luk, and J. D. Villasenor, "Hierarchical Segmentation for Hardware Function Evaluation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 1, pp. 103–116, Jan. 2009. DOI: 10.1109/TVLSI.2008.2003165.

[88] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 922–927, Jul. 2005. DOI: 10.1109/TC.2005.119.

[89] S. Sinha and W. Zhang, "Low-Power FPGA Design Using Memoization-Based Approximate Computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 8, pp. 2665–2678, Aug. 2016. DOI: 10.1109/TVLSI.2016.2520979.

[90] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2012, pp. 449–460. DOI: 10.1109/MICRO.2012.48.

[91] R. S. Amant *et al.*, "General-purpose code acceleration with limited-precision analog computation," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, Jun. 2014, pp. 505–516. DOI: 10.1109/ISCA.2014.6853213.

[92] T. Moreau *et al.*, "SNNAP: Approximate computing on programmable SoCs via neural acceleration," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2015, pp. 603–614. DOI: 10.1109/HPCA.2015.7056066.

[93] I. J. Chang, D. Mohapatra, and K. Roy, "A Priority-Based 6T/8T Hybrid SRAM Architecture for Aggressive Voltage Scaling in Video Applications," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 21, no. 2, pp. 101–112, Feb. 2011. DOI: 10.1109/TCSVT.2011.2105550.

[94] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture Support for Disciplined Approximate Programming," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII, New York, NY, USA: ACM, 2012, pp. 301–312. DOI: 10.1145/2150976.2151008.

[95] A. Ranjan, S. Venkataramani, X. Fong, K. Roy, and A. Raghunathan, "Approximate storage for energy efficient spintronic memories," in *Proceedings of the 52nd Annual Design Automation Conference*, ser. DAC '15, New York, NY, USA: Association for Computing Machinery, Jun. 2015, pp. 1–6. DOI: 10.1145/2744769.2744799.

[96] B. Salami, O. S. Unsal, and A. Cristal Kestelman, "Comprehensive Evaluation of Supply Voltage Underscaling in FPGA on-Chip Memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2018, pp. 724–736. DOI: 10.1109/MICRO.2018.00064.

[97] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM Refresh-power Through Critical Data Partitioning," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, New York, NY, USA: ACM, 2011, pp. 213–224. DOI: 10.1145/1950365.1950391.

[98] J. Lucas, M. Alvarez-mesa, M. Andersch, and B. Juurlink, "Sparkk : Quality-Scalable Approximate Storage in DRAM," in *The Memory Forum*, Minneapolis, 2014. [Online]. Available: https://www.cs.utah.edu/thememoryforum/lucas.pdf (visited on 04/08/2022).

[99] A. Raha, S. Sutar, H. Jayakumar, and V. Raghunathan, "Quality Configurable Approximate DRAM," *IEEE Transactions on Computers*, vol. 66, no. 7, pp. 1172–1187, Jul. 2017. DOI: 10.1109/TC.2016.2640296.

[100] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate Storage in Solid-state Memories," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46, New York, NY, USA: ACM, 2013, pp. 25–36. DOI: 10.1145/2540708.2540712.

[101] S. Ganapathy, G. Karakonstantis, A. Teman, and A. Burg, "Mitigating the impact of faults in unreliable memories for error-resilient applications," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, Jun. 2015, pp. 1–6. DOI: 10.1145/2744769.2744871.

[102] Y. Tian, Q. Zhang, T. Wang, F. Yuan, and Q. Xu, "ApproxMA: Approximate Memory Access for Dynamic Precision Scaling," in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, ser. GLSVLSI '15, New York, NY, USA: ACM, 2015, pp. 337–342. DOI: 10.1145/2742060.2743759.

[103] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th. USA: Addison-Wesley Publishing Company, 2010.

[104] B. Widrow, I. Kollar, and M.-C. Liu, "Statistical theory of quantization," *IEEE Transactions on Instrumentation and Measurement*, vol. 45, no. 2, pp. 353–361, Apr. 1996. DOI: 10.1109/19.492748.

[105] L. H. de Figueiredo and J. Stolfi, "Affine Arithmetic: Concepts and Applications," *Numerical Algorithms*, vol. 37, no. 1, pp. 147–158, Dec. 2004. DOI: 10.1023/B:NUMA.0000049462.70970.b6.

[106] D. May and W. Stechele, "An FPGA-based probability-aware fault simulator," in *2012 International Conference on Embedded Computer Systems (SAMOS)*, Jul. 2012, pp. 302–309. DOI: 10.1109/SAMOS.2012.6404190.

[107] J. F. Miller, *Cartesian Genetic Programming* (Natural Computing Series), J. F. Miller, Ed. Berlin, Heidelberg: Springer, 2011. DOI: 10.1007/978-3-642-17310-3_2.

[108] M. Hatamian and G. Cash, "A 70-MHz 8-bit/spl times/8-bit parallel pipelined multiplier in 2.5-/spl mu/m CMOS," *IEEE Journal of Solid-State Circuits*, vol. 21, no. 4, pp. 505–513, Aug. 1986. DOI: 10.1109/JSSC.1986.1052564.

[109] H. Jiang, F. J. H. Santiago, H. Mo, L. Liu, and J. Han, "Approximate Arithmetic Circuits: A Survey, Characterization, and Recent Applications," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2108–2135, Dec. 2020. DOI: 10.1109/JPROC.2020.3006451.

[110] J.-M. Muller, "Elementary Functions and Approximate Computing," *Proceedings of the IEEE*, pp. 1–14, 2020. DOI: 10.1109/JPROC.2020.2991885.

[111] J.-M. Muller, *Elementary Functions - Algorithms and Implementation*, J.-M. Muller, Ed. Boston, MA: Birkhäuser, 2016. DOI: 10.1007/978-1-4899-7983-4.

[112] F. de Dinechin and B. Pasca, "Designing Custom Arithmetic Data Paths with FloPoCo," *IEEE Design Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011. DOI: 10.1109/MDT.2011.44.

[113] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. 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*, ser. ESEC/FSE '11, New York, NY, USA: ACM, 2011, pp. 124–134. DOI: 10.1145/2025113.2025133.

[114] L. Lou, P. Nguyen, J. Lawrence, and C. Barnes, "Image Perforation: Automatically Accelerating Image Pipelines by Intelligently Skipping Samples," *ACM Trans. Graph.*, vol. 35, no. 5, 153:1–153:14, Sep. 2016. DOI: 10.1145/2904903.

[115] D. Citron and D. G. Feitelson, ""Look It Up" or "Do the Math": An Energy, Area, and Timing Analysis of Instruction Reuse and Memoization," in *Power-Aware Computer Systems*, ser. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, Dec. 2003, pp. 101–116. DOI: 10.1007/978-3-540-28641-7_8.

[116] D. G. Bailey, "Mapping Techniques," in *Design for Embedded Image Processing on FPGAs*, IEEE, 2011, pp. 107–154. DOI: 10.1002/9780470828519.ch5.

[117] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-Aware Intelligent DRAM Refresh," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 1–12, Jun. 2012. DOI: 10.1145/2366231.2337161.

[118] D. G. Bailey, "Point Operations," in *Design for Embedded Image Processing on FPGAs*, IEEE, 2011, pp. 155–197. DOI: 10.1002/9780470828519.ch6.

[119] M. Shafique, R. Hafiz, S. Rehman, W. El-Harouni, and J. Henkel, "Invited - Cross-layer approximate computing: From logic to architectures," in *Proceedings of the 53rd Annual Design Automation Conference*, Jun. 2016, pp. 1–6. DOI: 10.1145/2897937.2906199.

[120] D. Hernandez-Araya, J. Castro-Godínez, M. Shafique, and J. Henkel, "AUGER: A Tool for Generating Approximate Arithmetic Circuits," in *2020 IEEE 11th Latin American Symposium on Circuits Systems (LASCAS)*, Feb. 2020, pp. 1–4. DOI: 10.1109/LASCAS45839.2020.9069045.

[121] Intel Corporation, *Intel Quartus Prime Software Suite*. [Online]. Available: https://www.intel.de/content/www/de/de/software/programmable/quartus-prime/overview.html (visited on 04/08/2022).

[122] Siemens, *ModelSim*. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/modelsim/ (visited on 04/08/2022).

[123] Z. Vasicek, "Formal Methods for Exact Analysis of Approximate Circuits," *IEEE Access*, vol. 7, pp. 177 309–177 331, 2019. DOI: 10.1109/ACCESS.2019.2958605.

[124] B. S. Prabakaran, V. Mrazek, Z. Vasicek, L. Sekanina, and M. Shafique, "ApproxFPGAs: Embracing ASIC-Based Approximate Arithmetic Components for FPGA-Based Systems," in *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, ser. DAC '20, Virtual Event, USA: IEEE Press, Jul. 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218533.

[125] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. 85, pp. 2825–2830, 2011. [Online]. Available: https://dl.acm.org/doi/10.5555/1953048.2078195 (visited on 04/08/2022).

[126] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001. DOI: 10.1023/A:1010933404324.

[127] T. Hastie, R. Tibshirani, and J. Friedman, "Neural Networks," in *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, ser. Springer Series in Statistics, T. Hastie, R. Tibshirani, and J. Friedman, Eds., New York, NY: Springer, 2009, pp. 389–416. DOI: 10.1007/978-0-387-84858-7_11.

[128] *Scikit-learn RandomForestRegressor*. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor (visited on 04/08/2022).

[129] "Model assessment and selection," in *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, ser. Springer Series in Statistics, T. Hastie, R. Tibshirani, and J. Friedman, Eds., New York, NY: Springer, 2009, pp. 219–259. DOI: 10.1007/978-0-387-84858-7_7.

[130] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring Network Structure, Dynamics, and Function using NetworkX," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11–15. [Online]. Available: https://www.osti.gov/biblio/960616 (visited on 04/08/2022).

[131] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate Data Types for Safe and General Low-power Computation," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, New York, NY, USA: ACM, 2011, pp. 164–174. DOI: 10.1145/1993498.1993518.

[132] A. Rahimi, A. Marongiu, R. K. Gupta, and L. Benini, "A variability-aware OpenMP environment for efficient execution of accuracy-configurable computation on shared-FPU processor clusters," in *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Sep. 2013, pp. 1–10. DOI: 10.1109/CODES-ISSS.2013.6659022.

[133] M. Makni, S. Niar, M. Baklouti, and M. Abid, "HAPE: A high-level area-power estimation framework for FPGA-based accelerators," *Microprocessors and Microsystems*, vol. 63, pp. 11–27, Nov. 2018. DOI: 10.1016/j.micpro.2018.08.004.

[134] S. Andriani, H. Brendel, T. Seybold, and J. Goldstone, "Beyond the Kodak image set: A new reference set of color image sequences," in *2013 IEEE International Conference on Image Processing*, IEEE, Sep. 2013, pp. 2289–2293. DOI: 10.1109/ICIP.2013.6738472.

[135] Intel Corporation, *Intel Early Power Estimator*. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/support/support-resources/operation-and-testing/power/pow-powerplay.html (visited on 04/08/2022).

[136] Advanced Micro Devices, Inc, *Xilinx Power Estimator*. [Online]. Available: https://www.xilinx.com/products/technology/power/xpe.html (visited on 04/08/2022).

[137] Intel Corporation, *Power Analyzer*. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/support/support-resources/operation-and-testing/power/sof-qts-power.html (visited on 04/08/2022).

[138] 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*, vol. 13, no. 4, pp. 600–612, Apr. 2004. DOI: 10.1109/TIP.2003.819861.

[139] ISO/CIE 11664-4:2019, "Colorimetry — Part 4: CIE 1976 L*a*b* colour space," International Organization for Standardization, Geneva, CH, Standard, Jul. 2019. [Online]. Available: https://www.iso.org/standard/74166.html (visited on 04/08/2022).

[140] W. T. J. Chan, A. B. Kahng, S. Kang, R. Kumar, and J. Sartori, "Statistical analysis and modeling for error composition in approximate computation circuits," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, Oct. 2013, pp. 47–53. DOI: 10.1109/ICCD.2013.6657024.

[141] A. Ghasemazar and M. Lis, "Gaussian mixture error estimation for approximate circuits," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, Mar. 2017, pp. 302–305. DOI: 10.23919/DATE.2017.7927004.

[142] J. Castro-Godínez, S. Esser, M. Shafique, S. Pagani, and J. Henkel, "Compiler-driven error analysis for designing approximate accelerators," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2018, pp. 1027–1032. DOI: 10.23919/DATE.2018.8342163.

[143] D. Sengupta, F. S. Snigdha, J. Hu, and S. S. Sapatnekar, "An Analytical Approach for Error PMF Characterization in Approximate Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 1, pp. 70–83, Jan. 2019. DOI: 10.1109/TCAD.2018.2803626.

[144] S. Triantaphillidou and E. Allen, "Digital image file formats," in *The Manual of Photography (Tenth Edition)*, E. Allen and S. Triantaphillidou, Eds., Oxford: Focal Press, Jan. 2011, pp. 315–328. DOI: 10.1016/B978-0-240-52037-7.10017-1.

[145] S. Triantaphillidou, "Digital colour reproduction," in *The Manual of Photography (Tenth Edition)*, E. Allen and S. Triantaphillidou, Eds., Oxford: Focal Press, Jan. 2011, pp. 411–432. DOI: 10.1016/B978-0-240-52037-7.10023-7.

[146] Recommendation BT.709-6, "Parameter values for the HDTV standards for production and international programme exchange," International Organization for Standardization, Geneva, CH, Reccomendation, Jun. 2015. [Online]. Available: https://www.itu.int/rec/R-REC-BT.709-6-201506-I (visited on 04/08/2022).

[147] K. Nepal, Y. Li, R. I. Bahar, and S. Reda, "ABACUS: A technique for automated behavioral synthesis of approximate computing circuits," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2014, pp. 1–6. DOI: 10.7873/DATE.2014.374.

[148] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, Apr. 2002. DOI: 10.1109/4235.996017.

[149] K. Nepal, S. Hashemi, H. Tann, R. I. Bahar, and S. Reda, "Automated High-Level Generation of Low-Power Approximate Computing Circuits," *IEEE Transactions on Emerging Topics in Computing*, vol. 7, no. 1, pp. 18–30, Jan. 2019. DOI: 10.1109/TETC.2016.2598283.

[150] S. Xu and B. C. Schafer, "Exposing Approximate Computing Optimizations at Different Levels: From Behavioral to Gate-Level," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 11, pp. 3077–3088, Nov. 2017. DOI: 10.1109/TVLSI.2017.2735299.

[151] 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)*, Jun. 2019, pp. 1–6. DOI: 10.1145/3316781.3317781.

[152] J. Castro-Godínez, J. Mateus-Vargas, M. Shafique, and J. Henkel, "AxHLS: Design space exploration and high-level synthesis of approximate accelerators using approximate functional units and analytical models," in *Proceedings of the 39th International Conference on Computer-Aided Design*, ser. ICCAD '20, New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 1–9. DOI: 10.1145/3400302.3415732.

[153] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing* (Natural Computing Series), A. Eiben and J. Smith, Eds. Berlin, Heidelberg: Springer, 2015. DOI: 10.1007/978-3-662-44874-8.

[154] F.-M. De Rainville, F.-A. Fortin, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Enabling nimbler evolutions," *ACM SIGEVOlution*, vol. 6, no. 2, pp. 17–26, Feb. 2014. DOI: 10.1145/2597453.2597455.

[155] Reccomendation ITU-T T.871, "Information technology – Digital compression and coding of continuous-tone still images: JPEG File Interchange Format (JFIF)," International Organization for Standardization, Geneva, CH, Reccomendation, May 2011. [Online]. Available: https://www.itu.int/rec/T-REC-T.871-201105-I (visited on 04/08/2022).

[156] E. Reinhard, W. Heidrich, P. Debevec, S. Pattanaik, G. Ward, and K. Myszkowski, *High Dynamic Range Imaging*, Second. Burlington, MA, USA: Morgan Kaufmann, May 2010.

[157] E. Reinhard and K. Devlin, "Dynamic range reduction inspired by photoreceptor physiology," *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 1, pp. 13–24, Jan. 2005. DOI: 10.1109/TVCG.2005.9.

[158] H. R. Kang, "RGB Color Spaces," en, in *Computational Color Technology*, First, Bellingham, WA, USA: SPIE - The International Society for Optical Engineering, 2006, pp. 77–102. DOI: 10.1117/3.660835.ch6.

[159] IEC 61966-2-1:1999/AMD1:2003, "Multimedia Systems and Equipment - Colour Measurement and Management Part 2-1: Colour Management - Default RGB Colour Space - sRGB - Amendment 1," International Electrotechnical Commission, Geneva, CH, Standard, Jan. 2003. [Online]. Available: https://www.iso.org/standard/35883.html (visited on 04/08/2022).

[160] M. Stokes, M. D. Fairchild, and R. S. Berns, "Precision requirements for digital color reproduction," *ACM Trans. Graph.*, vol. 11, no. 4, pp. 406–422, Oct. 1992. DOI: 10.1145/146443.146482.

[161] H. R. Kang, "Three-Dimensional Lookup Table with Interpolation," en, in *Computational Color Technology*, First, Bellingham, WA, USA: SPIE - The International Society for Optical Engineering, 2006, pp. 151–182. DOI: 10.1117/3.660835.ch9.

[162] ARRI, *ARRI Color Tool*. [Online]. Available: https://www.arri.com/en/learn-help/learn-help-camera-system/tools/arri-color-tool (visited on 04/08/2022).

[163] D. G. Bailey, "Local Filters," in *Design for Embedded Image Processing on FPGAs*, IEEE, 2011, pp. 233–273. DOI: 10.1002/9780470828519.ch8.

List of Figures

List of Tables