12-2016

Approximate computing: An integrated cross-layer framework

Swagath Venkataramani
Purdue University

Graduate School Form 30
(Revised 08/14)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Swagath Venkataramani

Entitled Approximate Computing: An Integrated Cross-layer Framework

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

ANAND RAGHUNATHAN

CHENG-KOK KOH

KAUSHIK ROY

VIJAY RAGHUNATHAN

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

ANAND RAGHUNATHAN

Approved by Major Professor(s): _____

_____

Approved by: V. Balakrishnan 10/03/2016

Head of the Department Graduate Program Date

APPROXIMATE COMPUTING:

AN INTEGRATED CROSS-LAYER FRAMEWORK

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Swagath Venkataramani

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2016

Purdue University

West Lafayette, Indiana

கற்றது கைமண் அளவு, கல்லாதது உலகளவு

-- ஔவையார்

What you have learned is a mere handful;
What is yet to be learnt is the size of the world!

– Auvayar

ACKNOWLEDGMENTS

The true highlight of my PhD, more than the projects I worked on, was the people I worked with. I had the opportunity to collaborate with people from diverse technical backgrounds ranging from devices to algorithms, and the unique perspective that each brought forward was truly an enriching learning experience, for which I am ever grateful. I truly relish the relationships I have developed with my collaborators over the years, and wish to take it forward in the same spirit, as I take the next step in my profession career.

The single most reason behind all I have learnt and achieved, both professionally and personally, is my advisor Prof. Anand Raghunathan. He had undisputedly cared for my development and put my interests first in all decisions. I have always looked up to him and emulate him in all aspects of research, right from what problems to choose, what unique angles I as a researcher can bring in, where to spend my effort so that it is most impactful, and how best to present my work such that it is as intuitive to others as it is to me. The sheer experience of spending over half a decade with him has immensely helped me to organically develop as a researcher. I thank him all the flexibility and he is undoubtedly my superhero, the epitome of kindness, work ethic and intelligence. In the same breath, I thank all my teachers, right from my childhood, for who I am is a direct result of the knowledge they imparted.

My lab, Integrated Systems Laboratory, could not have been a more productive environment. My longevity at Purdue allowed me to work with some uniquely amazing people, seniors Vinay, Jacques, Vivek, Rangha, Yue Du, Younghyun and juniors Shankar, Ashish, Younghoon Shubham, Sanchari, Junshi, Neel, Arnab, and Sanjay. Needless to say, our relationships transcend beyond the walls of the MSEE 337. We always had our backs in thick and thin, and I enjoyed and learned from every *bit* (pun intended) of interaction. I truly care about each one of their successes, continue

to feel proud of their accomplishments and strive to imbibe their finest qualities in me. ISL is the prime example of a truly synergistic academic lab, its strength purely stemming from the mutual care and respect people show for each other.

I interacted significantly with Prof. Vijay Raghunathan and Prof. Kaushik Roy throughout my PhD. I thank them for their support and guidance at each important step in my development. I also worked closely with students from Nanoelectronics Research Labs, Embedded Systems Lab, Integrating Imaging Lab at Purdue. I equally cherish those relationships and look forward to continue working with them in future. I also thank those I collaborated with in the industry, specifically during my internships at Exa-scale computing research and parallel computing labs at Intel, and Sensing and energy research group at Microsoft. I thank my mentors for their support and trust in my abilities.

I spent most of my time outside lab at the badminton courts in Purdue. Badminton was so integral to me, and I thank the Purdue Badminton Club members and other badminton enthusiasts in the midwest region for all the fun and competition we had on and off court. I also thank my cricket buddies for the fun times we shared.

On the personal front, I am undeservedly fortunate to have met Vijay, Kaushik, Priyanka, Chandana, Shankar, Samyuktha and Tanmay at Purdue. The veracity in their characters demonstrated the true kinship I never felt over the years. They kept me honest and their care for me kept me unfazed. I attribute every smile I enjoyed and every pain I felt to them. I cannot understate the influence my roomate Ashiwan has had on me. I never felt a stranger when he was around, and eternal gratitude to him and Sanju. I also thank my friends, Leena, Anand and others at Bangalore for all the fond memories. I will be amiss not to thank my family in the US for their inspiration and incessant support.

Finally, I thank my parents, Chithra and Venkataramani, for unconditionally being there for me and imparting values that defines my very identity today. The following pages are a product their emboldened vision, unfettering hope, and countless scarifies, and for that I humbly dedicate what is rightly theirs to them.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

AxC	Approximate Computing
SALSA	Systematic Automatic Logic Synthesis of Approximate circuits
ALS	Approximate Logic Synthesis
QCC	Quality Constraint Circuit
ADC	Approximation Don't Cares
EXDC	External Don't Cares
SASIMI	Substitute-And-SIMplIfy
TS	Target Signal
SS	Substitute Signal
QPP	Quality Programmable Processor
QP-ISA	Quality Prgrammable Instuction Set Architecture
QP-uArch	Quality Programmable Micro-architecture
QUORA	Quality Programmable 1D/2D Vector Processor
APE	Approximate Processing Element
MAPE	Mixed Accuracy Processing Element
CAPE	Completely Accurate Processing Element
PScU	Precision Scaling Units
QCU	Quality Control Unit
SM	Streaming Memory
NN	Neural Networks
DLN	Deep Learning Networks
AxNN	Approximate Neural Networks
QCNPE	Quality Configurable Neural Processing Engine
NCU	Neuron Computation Unit

AFU	Activation Function Unit
RTL	Register Transfer Level
HW/SW	Hardware/Software
MAC	Multiply And Accumulate
SAD	Sum of Absolute Differences
DCT	Discrete Cosine Transform

ABSTRACT

Venkataramani, Swagath PhD, Purdue University, December 2016. Approximate Computing: An Integrated Cross-layer Framework. Major Professor: Anand Raghunathan.

We have witnessed a fundamental shift in the nature of workloads executed by computing platforms across the spectrum, from mobile and deeply-embedded devices to servers and data centers. Increasingly, computing platforms need to analyze, organize and search through large amounts of real-world data, intelligently interact with the physical world, be context-aware, and present more natural human interfaces. These tasks do not involve the computation of a golden answer or unique numerical result. Instead, they need to produce outputs that are good-enough or of sufficient quality. Such workloads possess *intrinsic application resilience*, or the ability to produce outputs of acceptable quality even when a large fraction of their computations are performed in an imprecise or approximate manner. Intrinsic application resilience offers an entirely new dimension along which computing platforms can be optimized. However, the design of computing platforms still continues to be guided by the dogma that every computation must be executed with the same strict notion of correctness. With the demand for computing performance growing unabated on the one hand, while traditional benefits due to technology scaling diminish on the other, it is important to leverage this new source of efficiency.

A new design approach, called *approximate computing* (AxC), leverages the flexibility provided by intrinsic application resilience to realize hardware or software implementations that are more efficient in energy or performance. Approximate computing techniques forsake exact (numerical or Boolean) equivalence in the execution of some of the application's computations, while ensuring that the output quality is

acceptable. While early efforts in approximate computing have demonstrated great potential, they consist of ad hoc techniques applied to a very narrow set of applications, leaving in question the applicability of approximate computing in a broader context.

The primary objective of this thesis is to develop an integrated cross-layer approach to approximate computing, and to thereby establish its applicability to a broader range of applications. The proposed framework comprises of three key components: (i) At the circuit level, systematic approaches to design approximate circuits, or circuits that realize a slightly modified function with improved efficiency, (ii) At the architecture level, utilize approximate circuits to build programmable approximate processors, and (iii) At the software level, methods to apply approximate computing to machine learning classifiers, which represent an important class of applications that are being utilized across the computing spectrum. Towards this end, the thesis extends the state-of-the-art in approximate computing in the following important directions.

Synthesis of Approximate Circuits. First, the thesis proposes a rigorous framework for the automatic synthesis of *approximate circuits*, which are the hardware building blocks of approximate computing platforms. Designing approximate circuits involves making judicious changes to the function implemented by the circuit such that its hardware complexity is lowered without violating the specified quality constraint. Inspired by classical approaches to Boolean optimization in logic synthesis, the thesis proposes two synthesis tools called SALSA and SASIMI that are general, *i.e.*, applicable to any given circuit and quality specification. The framework is further extended to automatically design *quality configurable circuits*, which are approximate circuits with the capability to reconfigure their quality at runtime. Over a wide range of arithmetic circuits, complex modules and complete datapaths, the circuits synthesized using the proposed framework demonstrate significant benefits in area and energy.

Programmable AxC Processors. Next, the thesis extends approximate computing to the realm of programmable processors by introducing the concept of *quality programmable processors* (QPPs). A key principle of QPPs is that the notion of quality is explicitly codified in their HW/SW interface *i.e.,* the instruction set. Instructions in the ISA are extended with quality fields, enabling software to specify the accuracy level that must be met during their execution. The micro-architecture is designed with hardware mechanisms to understand these quality specifications and translate them into energy savings. As a first embodiment of QPPs, the thesis presents a quality programmable 1D/2D vector processor QP-VEC, which contains a 3-tiered hierarchy of processing elements. Based on an implementation of QP-VEC with 289 processing elements, energy benefits upto 2.5× are demonstrated across a wide range of applications.

Software and Algorithms for AxC. Finally, the thesis addresses the problem of applying approximate computing to an important class of applications *viz.* machine learning classifiers such as deep learning networks. To this end, the thesis proposes two approaches—*AxNN* and *scalable effort classifiers*. Both approaches leverage domain-specific insights to transform a given application to an energy-efficient approximate version that meets a specified application output quality. In the context of deep learning networks, AxNN adapts backpropagation to identify neurons that contribute less significantly to the network's accuracy, approximating these neurons (*e.g.*, by using lower precision), and incrementally re-training the network to mitigate the impact of approximations on output quality. On the other hand, scalable effort classifiers leverage the heterogeneity in the inherent classification difficulty of inputs to dynamically modulate the effort expended by machine learning classifiers. This is achieved by building a chain of classifiers of progressively growing complexity (and accuracy) such that the number of stages used for classification scale with input difficulty. Scalable effort classifiers yield substantial energy benefits as a majority of the inputs require very low effort in real-world datasets.

In summary, the concepts and techniques presented in this thesis broaden the applicability of approximate computing, thus taking a significant step towards bringing approximate computing to the mainstream.

1. INTRODUCTION

Traditionally, computing platforms are viewed as calculators that execute tasks demanded by applications in a very precise manner. While the performance and efficiency of computing platforms have grown exponentially over the decades, their design continues to be guided by the principle that every computation they are tasked with is sacred and are carried out with a *strict (and unique) notion of correctness*. However, from a holistic application perspective, not all computations are equally important *i.e.,* a range of answers to underlying computations result in the same eventual application output. For example, consider the two tasks, shown in Figure 1.1, which are very similar to each other. In both cases, we wish to divide 433 by 21, but in the first case the output is compared to 20.4, while in the second, it is compared to 1. Computers today, due to their inability to understand the context in which the result

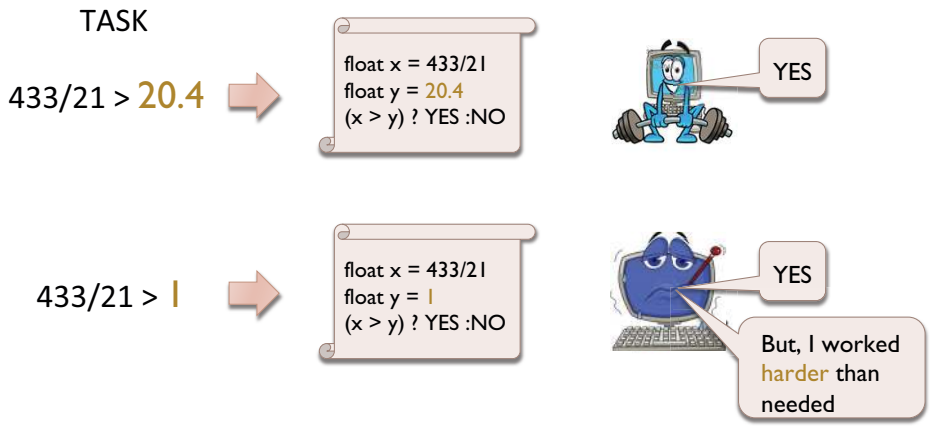


Fig. 1.1.: Illustration: Inefficiency in viewing computers as precise calculators

of the division operation is used, would expend the same effort to execute the tasks. However, as a high-level analogy, if the tasks were carried out by the human brain, it would find the second task to be much easier than the first, because it inherently

possesses the ability to compute results to the desired level of precision [1]. Hence, viewing computers as precise calculators clearly leads to significant in-efficiency and an overkill for many applications that computing platforms execute today.

1.1 Intrinsic Application Resilience

The landscape of computing applications has evolved over the years and we have witnessed a fundamental shift in the nature of workloads executed by computing platforms across the spectrum. With the ubiquity of the world wide web and the explosion in digital data of various forms, computing platforms in data centers and the cloud are used to analyze, interpret, and mine vast collections of raw data using semantic abstractions. At the other end of the spectrum, the need for intelligence and context-awareness in mobile and embedded devices implies that they too execute workloads that involve recognizing and interpreting data sensed from the physical world and their users. These workloads, which are of growing interest and expected to drive the usage of future computing platforms [2,3], do not expect a unique golden numerical answer. Rather, they are characterized by whether they produce an acceptable user experience, or results of sufficient quality. Even when a golden output is defined, the best known algorithms fall short of perfection (*e.g.*, for most real world recognition and classification problems, achieving 100% accuracy remains a distant objective). Hence, in their context, functional correctness is redefined from obtaining precise numerical answers to producing outputs that are "good-enough" to the end user (Figure 1.2).

These emerging workloads invariably exhibit significant *intrinsic application resilience*, which is broadly defined as the ability of applications to produce outputs of acceptable quality despite some of their underlying computations being executed in an imprecise or approximate manner. As illustrated in Figure 1.3, this intrinsic resilience, which is also shared by many prevalent application domains such as multimedia, graphics, and signal processing, stems from various factors [4–6]:

Fig. 1.2.: Emerging applications requiring good-enough answers

- The algorithms are designed to handle noisy real-world inputs, which as a consequence equips them to tolerate errors introduced in the intermediate computations.
- The computation patterns employed in these applications are typically iterative and statistical in nature allowing for errors introduced by approximations to attenuate or self-heal over time.
- A range of outputs are considered equivalent since no golden answer exists, or small deviations in the output cannot be perceived by users.

Recent studies [4] of intrinsic resilience in a suite of 12 emerging recognition, mining, and search applications revealed that on average, 83% of the runtime was spent in computations that were tolerant to errors in their outputs. This reaffirms the qualitative observation that a broad range of application domains exhibit significant intrinsic resilience.

The growth in demands being placed on computing platforms is expected to continue unabated, while on the other hand the benefits due to technology scaling continue to diminish. As illustrated in Figure 1.5, this growing *efficiency gap* is challenging designers of computing platforms and pushing them to innovate in hitherto

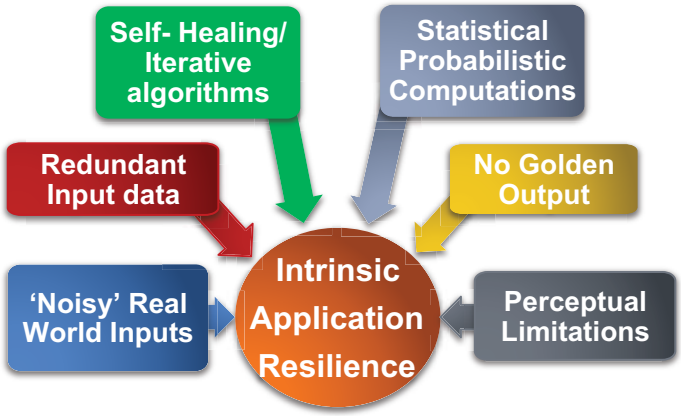


Fig. 1.3.: Intrinsic Application Resilience: Sources

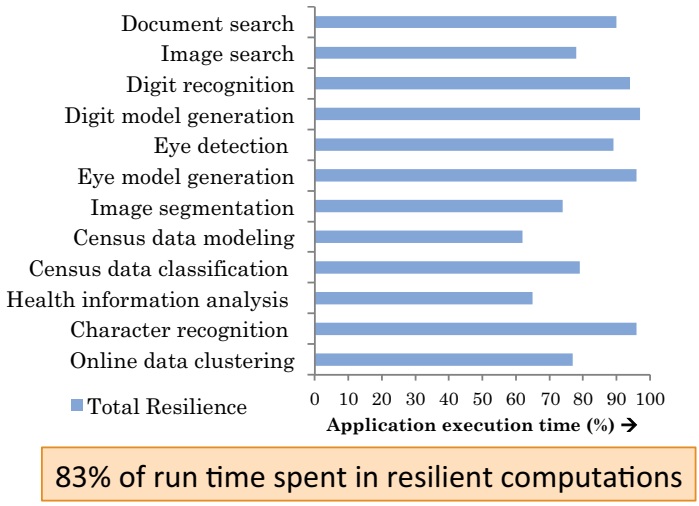


Fig. 1.4.: Fraction of execution time contributed by resilient computations

unexplored directions for improving the performance and energy efficiency. Some of the promising directions that are being explored include heterogeneous parallel architectures, near-threshold computing and accelerator-based computing. Optimizing designs by exploiting the intrinsic resilience of applications has the potential to become another dimension along which improvements in computing efficiency may be realized, since a large part of the growth in computing workloads can be attributed to application domains that possess significant resilience, including recognition, mining, synthesis, audio/video processing, data analytics, search, and graphics.

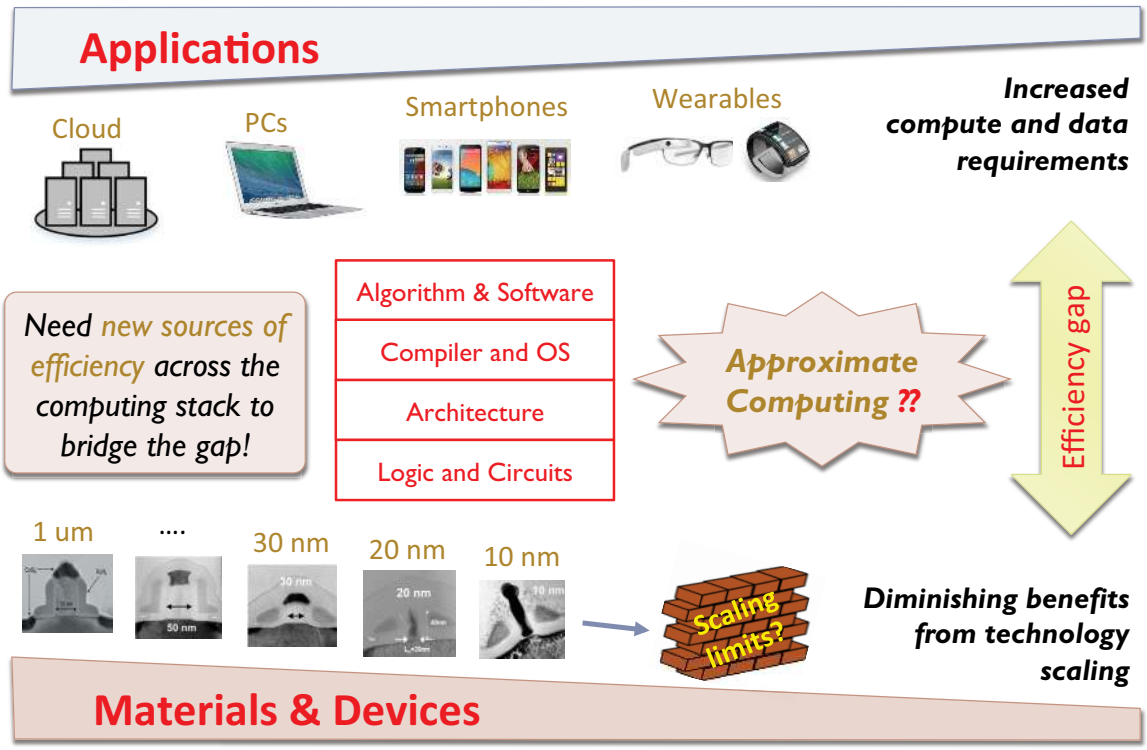


Fig. 1.5.: Efficiency gap in Computing

1.2 Approximate Computing

Approximate computing (AxC) is an emerging design approach that seeks to achieve new efficiencies in computing platforms by designing them such that they are capable of producing "good enough" results. Approximate computing platforms exploit the intrinsic resilience of applications by foregoing exactness or full correctness in the execution of selected computations, thereby improving performance or energy efficiency. Figure 1.6 illustrates the design principle behind approximate computing. Given the functional requirements of an application, the traditional design process involves several levels of design abstraction *viz.* developing the algorithm (or software), mapping the algorithmic computations to the desired hardware architecture, refining the architecture to circuit and layout to obtain the final implementation. One of the key invariants in the design process is that perfect equivalence (numerical or Boolean)

is maintained as the design progresses across the different levels of abstraction. As shown in Figure 1.6, approximate computing challenges this long-held dogma. In addition to the functional requirements of the application, the quality desired at the application output is also provided. Given these output quality specifications, approximations can be introduced in any of the design abstraction layers, provided the resultant degradation in output quality is acceptable. The key to the efficiency of approximate computing is to obtain a favorable a energy (or performance) *v.s.* quality trade-off *i.e.,* the benefits in energy is disproportionately large compared to the quality sacrificed in the process.

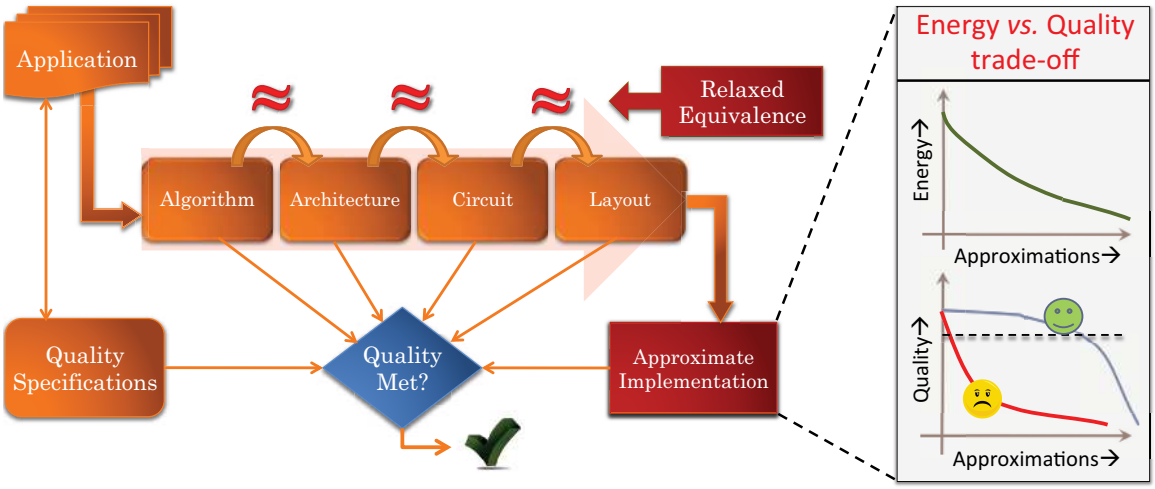


Fig. 1.6.: Approximate Computing: Design Principle

There has been significant interest in the area of approximate computing in recent years, and several techniques for approximate computing in hardware [6–13] or software [5,14–18] have been proposed. While these efforts have established the potential for significant improvements from approximate computing, they have invariably been *explored in an application-specific context* — the techniques are often ad hoc and applicable to specific applications, or the end result is often application-specific custom hardware. Consequently, two key questions that are frequently asked of approximate computing are "Is the approach applicable to a broader range of applications and domains?" and "Is it possible to amortize design effort by creating approximate com-

puting platforms that can be re-used across applications?" This dissertation aims to answer the above questions in the affirmative.

1.3 Thesis Contributions

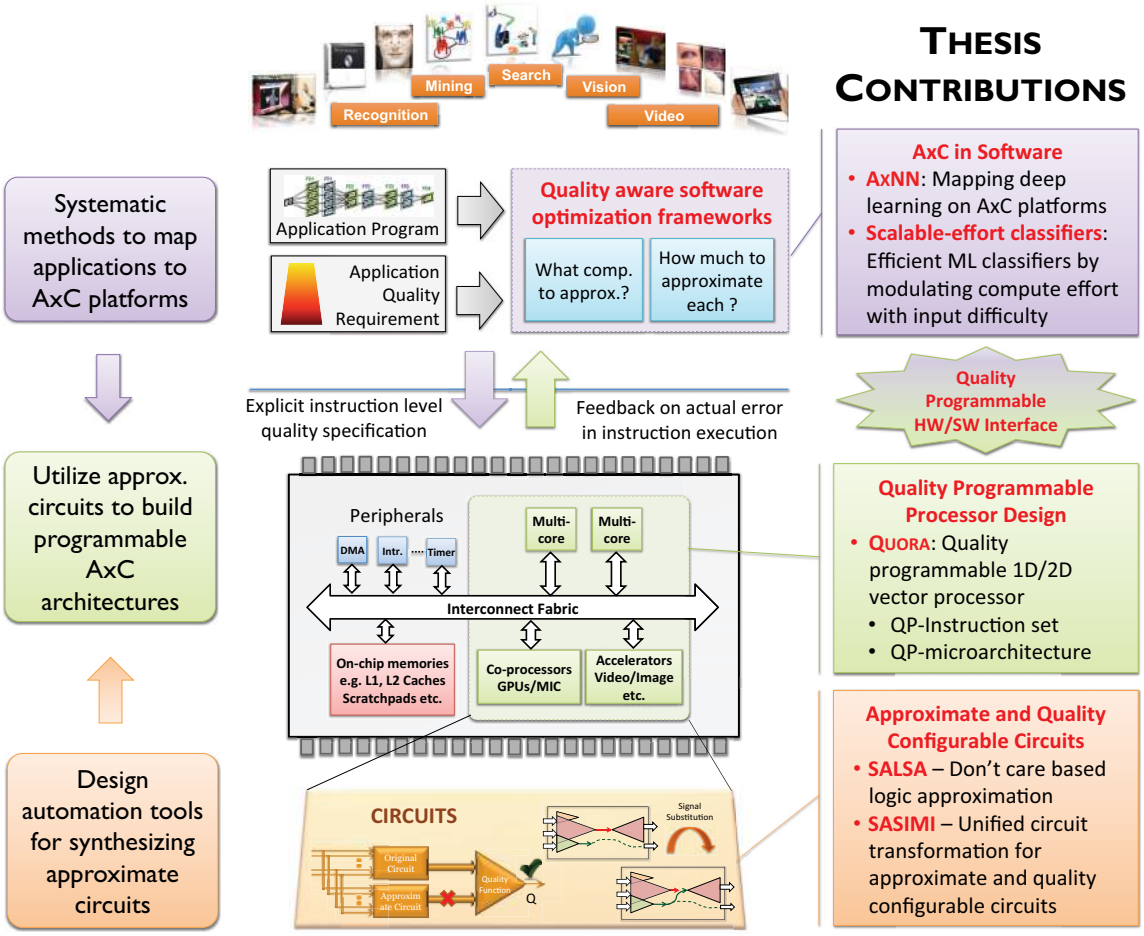


Fig. 1.7.: Contributions of the dissertation to approximate computing

To address the fundamental challenges associated with the broader adoption of approximate computing, this dissertation develops an integrated and systematic framework for approximate computing as outlined in Figure 1.7. The proposed framework comprises of three key components, which collectively involve developing approximate computing techniques at various layers of the computing stack. First, at the circuit

level, the thesis develops techniques to design approximate circuits that are highly efficient in performance, area and power. To enable generality and scalability, the thesis embodies the fundamental approximate circuit design principles into synthesis frameworks that can automatically generate "correct-by-construction" approximate versions for any given circuit, and any desired accuracy bound specified at the circuit outputs. Next, at the architecture level, the approximate circuits are used to build approximate computing architectures that yield a favorable trade-off between efficiency and application output quality. Towards this end, the thesis investigates how approximate computing can be best leveraged in the context of programmable processors. Finally, at the software level, the thesis develops methodologies to systematically identify resilient computations within an application and map them to approximate computing platforms. An overview of the important contributions of the dissertation are described below.

1.3.1 Approximate and Quality Configurable Circuits: Design and Synthesis

Approximate circuits are the basic hardware building blocks of approximate computing platforms. Approximate circuits are highly efficient hardware implementations that realize a slightly modified logic function compared to the original specifications within a specified quality constraint. Most research efforts in the area of approximate circuits can be summarized as manual designs of simple arithmetic circuits such as adders [19–22] and multipliers [23]. However, the broader adoption of approximate circuits requires a systematic methodology to design approximate implementations for any arbitrary circuit. Moreover, it is critical that such a methodology enable the generation of "correct-by-construction" approximate circuits that are guaranteed to satisfy designer-specified quality constraints, which is often not the case for the aforementioned manual designs.

Towards this objective, the thesis presents two synthesis methodologies *viz.* SALSA and SASIMI for the automatic design of approximate circuits. We further extend the methodology to synthesize quality configurable circuits, which possess the additional capability to reconfigure their quality at runtime. A brief summary of the methodologies are presented below.

SALSA

The first methodology SALSA extends the classical don't care based optimization approach for Approximate Logic Synthesis (ALS). The key hallmark of SALSA is the rigorous formulation of the problem of ALS into an equivalent traditional logic synthesis problem, thereby allowing the capabilities of existing synthesis tools to be fully utilized for logic approximation. SALSA achieves this by forming a virtual Quality Constraint Circuit (QCC) that encodes the quality constraints using logic functions called *Q-functions*. It then captures the flexibility engendered by the relaxed quality specifications as *Approximation Don't Cares* (ADCs), which are used for circuit simplification using traditional don't care based optimization techniques.

SASIMI

The second methodology SASIMI, introduces a new circuit transformation called Substitute-and-Simplify, for approximate circuit design. The key insight behind SASIMI is to identify signal pairs in the circuit that assume the same value with high probability, and substitute one for the other. While these substitutions introduce functional approximations, if performed judiciously, they result in some logic to be eliminated from the circuit while also enabling downsizing of gates on critical paths (simplification), resulting in significant power savings. SASIMI performs the substitution and simplification iteratively, while ensuring that a user-specified quality constraint is satisfied. SASIMI is extended to perform automatic synthesis of quality configurable circuits that can dynamically operate at different accuracy levels depend-

ing on application requirements. It is worthy to note that the quality configurable circuits synthesized by SASIMI do not incur any energy overheads even during the accurate mode of operation.

The synthesis tools, SALSA and SASIMI, were prototyped and utilized to synthesize approximate and quality configurable versions of a wide range of circuits comprised of arithmetic building blocks (adders, multipliers, MAC), ISCAS benchmarks and entire datapaths (DCT, FIR, IIR, SAD, FFT Butterfly, Euclidean distance), demonstrating scalability and significant improvements in area (1.1X to 1.85X for tight error constraints, and 1.2X to 4.75X for relaxed error constraints) and energy (1.15X to 1.75X for tight error constraints, and 1.3X to 5.25X for relaxed error constraints).

1.3.2 Programmable Approximate Computing Architectures

The thesis extends approximate computing to realm of programmable processors by introducing the concept of Quality Programmable Processors (QPPs). In QPPs, as shown in Figure 1.7, the conventional HW/SW interface, which allows specification of just the operation and the operands, is enhanced to explicitly embody the notion of quality. The ISA of a quality programmable processor contains instructions associated with quality fields to specify the accuracy level that must be met during their execution. It thus empowers software with the ability to specify not just what the operation is but also how significant it is in the context of the application. This ability to control the accuracy of instruction execution greatly enhances the scope of approximate computing, allowing it to be applied to larger parts of programs. The micro-architecture of a quality programmable processor contains hardware mechanisms that translate the instruction-level quality specifications into energy savings. Additionally, it may expose the actual error incurred during the execution of each instruction (which may be less than the specified limit) back to software.

As a first embodiment of quality programmable processors, the thesis presents the design of QUORA, an energy efficient, quality programmable vector processor. QUORA utilizes a 3-tiered hierarchy of processing elements that provide distinctly different energy *vs.* quality trade-offs, and uses hardware mechanisms based on precision scaling with error monitoring and compensation to facilitate quality programmable execution. We evaluate an implementation of QUORA with 289 processing elements in 45nm technology. The results demonstrate that leveraging quality-programmability leads to 1.05×-1.7× savings in energy for virtually no loss (<0.5%) in application output quality, and 1.18×-2.1× energy savings for modest impact (<2.5%) on output quality.

1.3.3 Software and Algorithms for Approximate Computing

At the software level, given an application and an application output quality requirement, the key challenge is to identify which computations to approximate and by how much. This requires quality-aware software optimization frameworks to identify resilient computations and quantitatively evaluate how approximations to these computations impact the application output. The dissertation addresses this problem in the context of an important class of applications *viz.* machine/deep learning classifiers. To this end, the thesis proposes the following approximation frameworks that leverage domain-specific insights to systematically transform a given application into its approximate version.

AxNN: Approximate Neural Networks

Large-scale neural networks or Deep Learning Networks (DLNs) have become popular due to their state-of-the-art performance on a wide range of machine learning problems. One of the key challenges with deep learning networks is their high computational complexity. To improve the efficiency of DLNs while preserving their functional performance, the thesis proposes a method to transform any given neural

network (NN) into an Approximate Neural Network (AxNN). This is performed by (i) adapting the backpropagation technique, which is commonly used to train these networks, to quantify the impact of approximating each neuron to the overall network quality (*e.g.,* classification accuracy), and (ii) selectively approximating those neurons that impact network quality the least. Further, leveraging the key observation that training is a naturally error-healing process, the network is incrementally retrained with the approximations in-place, reclaiming a significant portion of the quality ceded by approximations. We evaluated the proposed approach by constructing AXNNs for 6 recognition applications (ranging in complexity from 12-47,818 neurons and 160-3,155,968 connections). Our results demonstrate 1.14×-1.92× energy benefits for virtually no loss ($< 0.5\%$) in output quality, and even higher improvements (upto 2.3×) when some loss (upto 7.5%) in output quality is acceptable.

Scalable Effort Classifiers

Scalable effort classifiers are a new approach to optimizing the energy efficiency of supervised machine-learning classifiers. Its efficiency stems from the observation that the inherent classification difficulty varies widely across inputs in real-world datasets; only a small fraction of the inputs truly require the full computational effort of the classifier, while the large majority can be classified correctly with very low effort. Yet, state-of-the-art classification algorithms expend equal effort on all inputs, irrespective of their difficulty. To address this inefficiency, scalable effort classifiers dynamically adjust their computational effort depending on the difficulty of the input data, while maintaining the same level of accuracy. Scalable effort classifiers are constructed by utilizing a chain of classifiers with increasing levels of complexity (and accuracy). Scalable effort execution is achieved by modulating the number of stages used for classifying a given input. Every stage in the chain contains an ensemble of biased classifiers, where each biased classifier is trained to detect a single class more accurately. The degree of consensus between the biased classifiers' outputs is used

to decide whether classification can be terminated at the current stage or not. Thus any given classification algorithm can be transformed into a scalable effort chain. We build scalable effort versions of 8 popular recognition applications using 3 different classification algorithms. Our experiments demonstrate that scalable effort classifiers yield 2.79× reduction in average operations per input, which translates to 2.3× and 1.5× improvement in energy for hardware and software implementations, respectively.

In summary, the dissertation outlines an integrated framework for approximate computing that includes automatic frameworks to synthesize approximate circuit blocks, a programmable architecture along with its HW/SW interface that explicitly embodies the notion of quality, and finally software methodologies to systematically identify resilient computations within an application in order maximize the benefits for a desired output quality.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 describes the synthesis methodologies proposed for approximate and quality configurable circuit design. Chapter 3 details the concept of quality programmable processors, and the key issues involved in their design. Chapter 4 describes AxNN, the approach proposed to systematically identify and approximate resilient computations in deep learning networks. Chapter 5 outlines the concept of scalable effort classifiers and the methodology employed to construct scalable effort versions of any given classification algorithm. Chapter 6 presents the previous related efforts in the area of approximate computing and places the contributions of this thesis in their context. Finally, Chapter 7 provides a summary and outlines the key directions for future research.

2. DESIGN AND SYNTHESIS OF APPROXIMATE AND QUALITY CONFIGURABLE CIRCUITS

2.1 Introduction

Approximate circuits, or circuits that have lower hardware complexity (switched capacitance, leakage, and critical path), while evaluating the required function within a desired accuracy, are key ingredients in the design of an approximate computing system. Given the golden specifications of a circuit and a quality constraint that denotes the type and amount of error that the implementation can accommodate, the objective of the approximate circuit design process, as illustrated in Figure 2.1, is to make judicious changes to the function to be implemented such that it translates to a more efficient implementation, while differing from the original specification in a manner bounded by the specified quality constraint. The quality constraint is typically dictated by the application based on the context in which the output of the circuit is used.

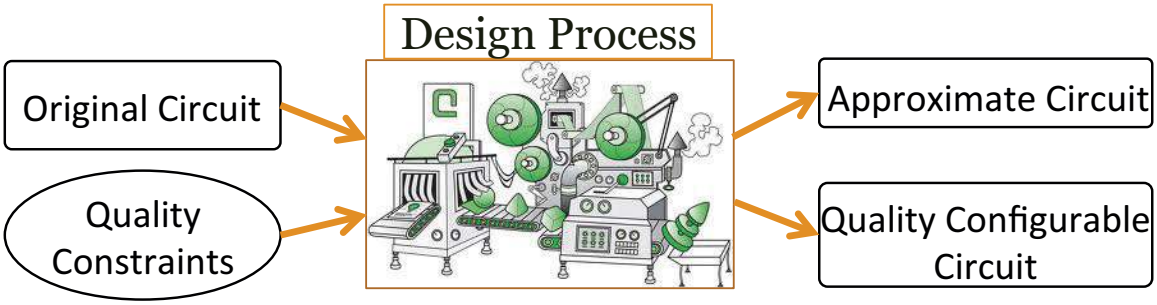


Fig. 2.1.: Approximate and quality configurable circuit design

In many applications, the degree of resilience often varies across computations depending on the application context or the dataset being processed [4, 8]. For example, consider the JPEG image compression application, shown in Figure 2.2, in which

each 8x8 block of the image is converted to its frequency domain representation using 2-D Discrete Cosine Transform (DCT). It is well known that the output of the JPEG application is impacted the most by the DC component at the top-left corner of the image compared to to other high frequency components. Now, if the computations corresponding to each component are mapped to the same underlying hardware, then it is necessary to operate the hardware with different accuracies depending on the significance of the component. In such scenarios, it is desirable to construct a more sophisticated variant of approximate circuits, called *quality-configurable circuits*, which are capable of reconfiguring to adapt their accuracy at run-time. Quality configurable circuits typically contain additional inputs to indicate the current quality requirement and the circuits are embodied with the capability to dynamically adapt their accuracy and energy consumption accordingly. The quality constraint during their design process comprises of a series of quality levels that are desired during operation.

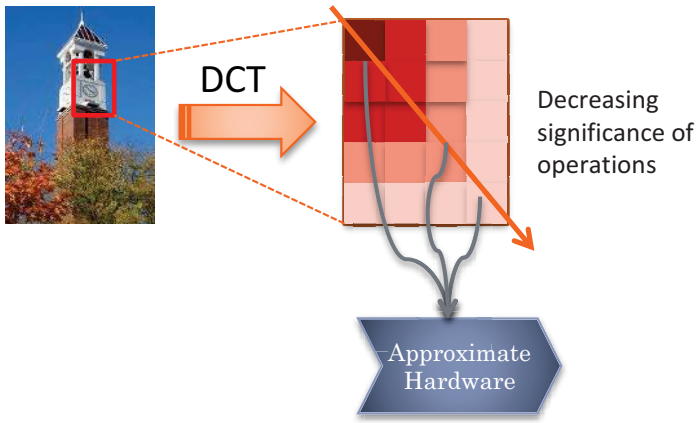


Fig. 2.2.: Need for quality configurable circuits

Traditionally, when it comes to the design of approximate circuits, there have been two major schools of thought: (i) Timing approximation, where the circuit is subject to voltage over-scaling resulting in timing errors [24, 25], and (ii) Functional approximation, where the circuit realizes a slightly different logic function than specified, resulting in a more efficient implementation [19–23, 26]. Currently, the design

of approximate circuits is (perhaps too much of) an art, requiring significant manual effort. Previous efforts largely consist of manual designs of simple arithmetic circuits such as adders [19–22, 26] and multipliers [23]. However, a key requirement for their mainstream adoption is to develop systematic design techniques and synthesis tools that are general and scalable to arbitrary circuits and quality constraints. Ideally, tools should:

- Allow designers to simply specify the circuit and quality constraint, relieving them from the burden of how to perform the approximation.
- Generate "correct-by-construction" approximate and quality configurable circuits that are *guaranteed* to satisfy the imposed quality constraints.
- Effectively translate the flexibility engendered by the quality constraints into improvements in performance or energy consumption.

This chapter presents a rigorous framework for the automatic synthesis of approximate and quality configurable logic circuits that achieves the aforementioned objectives. Traditional Boolean optimization approaches in logic synthesis can be broadly classified into two categories [27]. One class of techniques performs local optimizations in the circuit by identifying don't care conditions on the circuit nodes [28, 29], while the other introduces perturbations in the circuit, *e.g.,* by adding wires, in-order to potentially enable the removal of redundant logic [30]. Drawing inspiration from such classical approaches, this chapter presents two approaches *viz.* SALSA [31] and SASIMI [32], for the automatic synthesis of approximate circuits. SALSA extends the classical approach of implicit don't care based optimization through *approximation don't cares*, an entirely new class of don't cares that represent the functional flexibility afforded by the specified quality constraint. SASIMI, on the other hand, adopts the circuit transformation approach and introduces *substitute-and-simplify* as a new transformation to generating approximate and quality configurable circuits.

The rest of the chapter is organized as follows. Section 2.2 presents an overview of the quality metrics commonly used in evaluating approximate circuits. A detailed

description of the two design approaches SALSA and SASIMI and the methodology adopted in their respective implementations are then presented in the Sections 2.3 and 2.4. Finally, Section 2.5 provides a summary and concludes the chapter.

2.2 Quality Metrics

Before describing the approximate design techniques, we present an overview of the quality metrics that are typically employed in constraining the quality of the approximate circuit. As described in Section 2.1, quality metrics provide a bound on the type and amount of error that can be introduced in the implementation during the process of approximating the circuit function. They are typically a function of the original (O_{orig}) and approximate (O_{approx}) circuit implementations. Quality metrics can be classified into three broad categories‘

2.2.1 Metrics Constraining the Magnitude of Error

The first class of quality metrics constrain the quantity or magnitude of error at the output of the circuit. The bound in the magnitude of error may be either absolute *i.e.* true for every input to circuit or statistical over all possible circuit inputs. Some of the most common quality metrics belonging to both the categories are described below.

Maximum Error Magnitude: The maximum error ($MaxErr$) metric, shown in Equation 2.1, bounds the absolute difference in magnitude between the outputs of the original and approximate circuits to be less than a specified threshold.

$$MaxErr = MAX_{\forall inputs}(|O_{acc} - O_{approx}|) \tag{2.1}$$

Relative Error Magnitude: The relative error ($RelErr$) metric, shown in equation 2.2, constraints the ratio of the original and approximate circuit outputs to differ from 1 by at most a certain margin.

$$RelErr = \left| \frac{O_{approx}}{O_{orig}} \right| \tag{2.2}$$

Average Error Magnitude: The average error ($AveErr$) metric, shown in Equation 2.3, bounds the absolute difference in magnitude between the original and approximate circuits, averaged over all possible inputs.

$$AveErr = \frac{\sum_{\forall inputs} |O_{orig} - O_{approx}|}{Total\ number\ of\ Inputs} \tag{2.3}$$

Mean Squared Error Magnitude: As shown in Equation 2.4, the mean squared error ($MSErr$) metric constrains the mean of the squared difference in magnitude between the outputs of the original and approximate circuits over all possible inputs to be less than a specified threshold.

$$AveErr = \frac{\sum_{\forall inputs} (O_{orig} - O_{approx})^2}{Total\ number\ of\ Inputs} \tag{2.4}$$

Unidirectional Error: In addition to constraining the absolute magnitude of error, unidirectional quality metrics place a restriction on the direction in which error occurs to be either positive or negative. Unidirectional variants can be conceived for each of the error magnitude based quality metrics described above.

2.2.2 Metrics Constraining the Frequency of Error

The second class of quality metrics constrain the frequency of error *i.e.* the number of inputs for which the circuit can an incorrect value. These metrics are particularly useful when the output of the circuit does not represent a numerical value. Few examples of quality metrics in this class are described below.

Error Probability: The most prominent quality metric in this class is the error probability ($ErrProb$) metric shown in Equation 2.5. It is defined as the fraction of inputs vectors for which the approximate circuit output differs from the original circuit.

$$ErrProb = \frac{Total\ Inputs\ for\ which\ O_{orig} \neq O_{approx}}{Total\ number\ of\ Inputs} \tag{2.5}$$

Bit Error Probability: The bit error probability ($BitErrProb$) metric, shown in Equation 2.6, is similar to the error probability metric, but in this case, the error probabilities of individual output bits are constrained separately.

$$BitErrProb^i = \frac{Total\ Inputs\ for\ which\ O^i_{orig} \neq O^i_{approx}}{Total\ number\ of\ Inputs} \tag{2.6}$$

2.2.3 Composite Metrics

Composite quality metrics are a combination of both the above classes, wherein the approximate circuits are constrained in both the magnitude and the frequency of error.

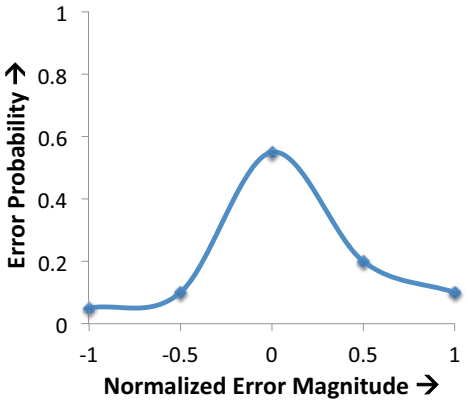


Fig. 2.3.: Error Probability Distribution

These metrics are commonly represented as an error probability distribution, shown in Figure 2.3, in which the X-axis represents the magnitude of error and the Y-axis provides the probability with which error of a given magnitude can occur in the approximate implementation.

2.3 SALSA: Don't Care based Logic Approximation

The first approximate design technique SALSA extends the concept of traditional don't cares to approximate logic synthesis. Starting with an RTL description of the exact circuit and a quality constraint that specifies the type and amount of error tolerable in the implementation, SALSA automatically synthesizes a functionally approximate version of the circuit that adheres to the pre-specified error constraints. The key steps involved in the proposed SALSA methodology are described below. First, SALSA constructs a virtual *Quality Constraint Circuit* (QCC) that encodes the specified quality constraints using logic functions called Q-functions. This enables SALSA to enforce the quality constraints during synthesis. Next, using the QCC, SALSA identifies Approximation Don't Cares (ADCs), an entirely new class of don't cares that are borne out of the quality specifications. These ADCs are then used for circuit simplification by leveraging standard don't care based optimization methodologies.

Thus SALSA rigorously reformulates the problem of Approximate Logic Synthesis (ALS) and maps it into a traditional logic synthesis problem. The problem formulation and the solution approach adopted in SALSA beget the following advantages:

- The proposed methodology provides an inherent guarantee that the specified bounds are never transgressed, thus enabling synthesis of correct-by-construction approximate circuits.

- The transformations are completely independent of the target error metric as well as the circuit considered for approximation. In essence, this decouples the synthesis procedure from the error metric, making this approach flexible and general. A variety of errors metrics can thus be specified based on the application requirements.

- Additionally, by virtue of transforming and mapping ALS to a traditional logic synthesis problem, existing off-the-shelf logic synthesis tools could just be reused for approximate circuit synthesis. This obviates the need for developing

a custom tool for ALS, thus lowering the barrier to adoption. Further, this widens the scope of approximations that can be effected on the circuits, since the entire power of existing logic optimization algorithms can be leveraged.

SALSA is prototyped using two different logic synthesis tools *viz.* SIS [33] and Synopsys Design Compiler [34], thereby demonstrating generality and used it to synthesize a range of arithmetic circuits and datapaths. The approximate circuits synthesized by SALSA achieve significant reductions in area and power.

2.3.1 Preliminaries and Approach

The problem statement for approximate logic synthesis could be articulated as follows. Given the description of a logic circuit and a constraint on the errors that could be tolerated, the synthesis procedure should identify avenues for logic simplification and generate a functionally approximate version of the that satisfies the pre-defined error bounds. The following sections describes the approach used in SALSA to accomplish this objective.

Quality constraint circuit

Figure 2.4 shows the Quality Constraint Circuit (QCC) that is used in SALSA to formulate the problem of approximate synthesis. The QCC is composed of three major blocks *viz.* the *Original circuit*, the *Approximate circuit* and the *Quality function* (Q-function). The original circuit block contains a structural description of the circuit that needs to be approximated and the error constraints that are to be satisfied are encoded into the Q-function. From the problem definition, both these blocks are available as inputs to SALSA. The task of SALSA is to synthesize the approximate circuit, so that the constraints set in the Q-function are never violated.

The inputs to the QCC are the primary inputs of the circuit considered for approximation. The output of the QCC is a single bit Q that indicates whether the constraints encoded into the Q-function are satisfied. The Q-function takes outputs

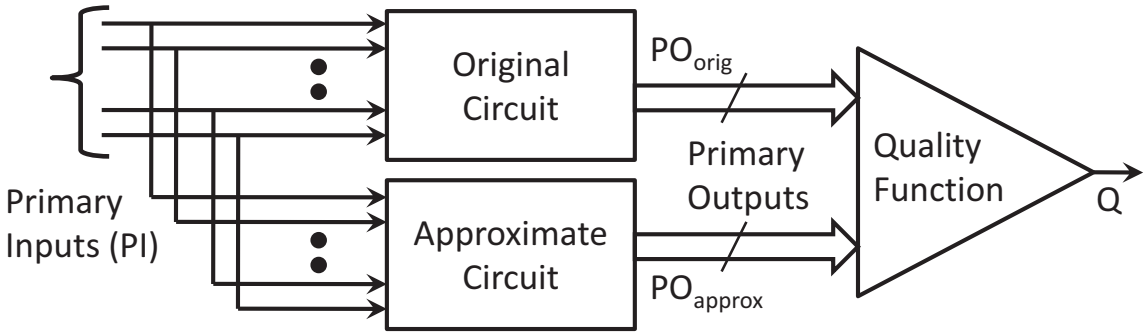


Fig. 2.4.: Quality constraint circuit

from both the original circuit PO_{orig} and approximate circuit PO_{approx} and decides if the quality constraints are satisfied. A Q output of logic '1' means that the approximate circuit conforms to the imposed quality bounds whereas a logic '0' output indicates a transgression. Thus, the QCC determines the legitimacy of the approximate circuit. From a functional viewpoint, for the approximate circuit to be valid, we need to ensure that Q evaluates to '1' for all possible input combinations. Stated otherwise, the QCC with the synthesized approximate circuit should evaluate to a tautology. At all times during the approximate synthesis process, SALSA preserves this invariant.

Quality function

As mentioned earlier, the Q-function takes in outputs from the original and approximate circuits and generates a single bit output indicating if the quality constraints are satisfied. In a circuit with M primary outputs, the Q-function maps 2M inputs into a one bit output. For example, consider the *maximum error magnitude* metric described in Section 2.2, in which the approximate output is constrained to differ from the correct output by no more than a specified value. In this case, the Q-function, as shown in Equation 2.7, is composed of a subtractor that evaluates

the difference between the original and approximate circuits outputs followed by a comparator circuit the compares the difference with a pre-determined threshold.

$$Q = \left(|PO_{orig} - PO_{approx}| \leq K \right) \quad ? \quad 1 : 0 \tag{2.7}$$

Another example is the *relative error* metric for which the Q-function, as shown in Equation 2.8, is comprised of a divider that computes the ratio of the approximate and original circuit outputs followed by two comparators.

$$Q = \left(1 - K \leq \frac{PO_{approx}}{PO_{orig}} \leq 1 + K \right) \quad ? \quad 1 : 0 \tag{2.8}$$

Thus, in SALSA, any quality metric that could be expressed as a Boolean function of the original and approximate circuit output bits could be specified as the Q-function.

Approximation don't cares

We next describe the strategy used in SALSA to transform the ALS problem into a traditional logic synthesis problem. In the QCC, the primary outputs of the original and approximate circuit represent internal nodes. We know that the outputs of the approximate circuit PO_{approx} are valid provided that they do not cause the value at Q to evaluate to '0' for any input value. In other words, we could functionally modify the approximate circuit if the change can never affect the value of Q.

In multi-level logic synthesis, the Observability Don't Cares (ODCs) of a node in a logic circuit can be defined as the set of input values for which the primary outputs of the circuit remain insensitive to the node's output [27]. These input combinations can be used to simplify the node because they do not affect the primary outputs of the circuit.

Applying this concept in our scenario, finding the observability don't cares at a bit of PO_{approx} (which is an internal signal in the QCC) gives us the set of primary input values for which Q is insensitive to an output of the approximate circuit. As

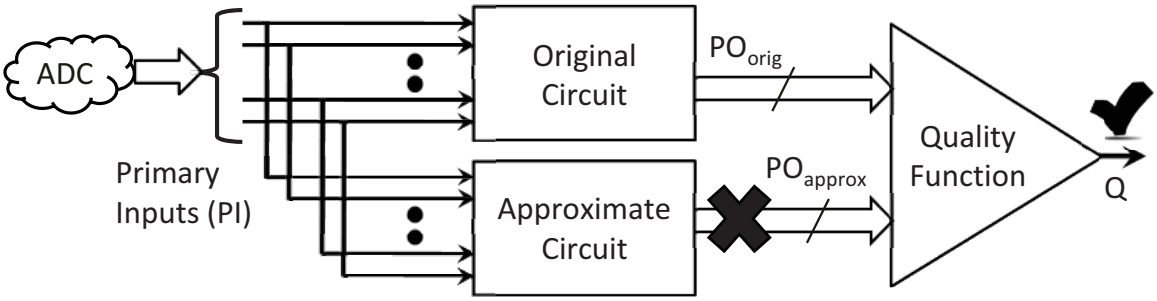


Fig. 2.5.: Approximation don't cares

shown in Figure 2.5, when the ODCs occur at inputs, the approximate output can be incorrect as it does not affect Q. SALSA uses this information to aid in approximating the circuit, and by virtue of their special significance these ODCs are termed as *Approximation Don't Cares* (ADCs) of the circuit.

The question that remains is how we could make use of the ADCs to approximate the circuit. We know that External Don't Cares (EXDCs) of an output in a circuit are the set of primary input combinations for which that primary output is a don't care. In our case, if the approximate circuit block is looked at in isolation, the ADCs for a given bit of PO_{approx} could be considered as the external don't cares for that output. Therefore, by setting these input combinations (ADCs) as EXDCs of an output in the approximate circuit, we could legally simplify or (in our context) approximate the cone of logic generating that output using standard don't care based synthesis techniques [28–30].

Selection of high effort ADCs

The approach described above can be utilized to approximate any circuit for which the quality metric is entirely expressed in form of a logic circuit within the Q-function of the QCC. This is possible in the case of quality metrics, such as maximum error magnitude and relative error, in which only the quantity or magnitude of error at the circuit output is constrained. However, as described in Section 2.2, the quality metric

may also additionally bound the number of inputs for which the circuit can produce an erroneous output. For example, consider the *error probability* metric. In this case, quality is defined as the fraction of inputs that result in an incorrect output *i.e.* the actual magnitude of error at the output is unconstrained but rather the number of inputs that can produce an error is bounded. Thus in cases where quality is defined over many inputs to the circuit, it cannot be completely expressed as a logic circuit in the QCC. In order to be able to handle such quality metrics in SALSA, we adopt the following approach. We make a key observation that only the input vectors the are defined as an ADC can result in an incorrect output after the circuit is approximated. Therefore, bounding the number of ADCs specified for a given output automatically constrains the fraction of inputs for which the output can be incorrect. Hence, based on the probability of error, a subset of vectors is chosen from the ADC set previously computed and used as EXDCs during circuit simplification.

The question that remains to be answered is that which vectors from the ADC set should be chosen to be designated as EXDCs. For this purpose, we define *effort* of a input vector as the number of wires in the circuit that the input sets to a non-controlling value. The tenet behind this definition is that, if a given input vector sets most wires in the circuit to a non-controlling value, then it potentially uses a large amount of logic in the circuit for generating the output. Hence selecting such high effort input vectors as ADCs has the potential to result in better logic simplification. Note that in the case of the error probability metric, since no bound on the magnitude of error is imposed, the Q-function is unity. This means that every input vector is an ADC and a subset from entire input space could to be chosen for approximation. However, for other composite metrics where both the magnitude and probability of error at the output is bounded, the search space for high effort vectors is limited to the ADC set computed using QCC.

Iterative simplification

In the procedure described above, it is important to note that when one output is being approximated, the functionality of all other outputs remain unaffected. This is because the ADCs, by definition, are specific to a given output bit and do not influence other outputs in any way. However, there could be avenues for approximation in the cones of logic of other output bits and hence this process of approximation should be repeated for all output bits. After each approximation, the QCC setup is updated with the latest available approximate circuit before computing the ADCs for the next output bit.

In summary, the key steps in SALSA are as follows:

- Form QCC and compute ADCs at an output bit of the approximate circuit.
- If the probability of error is additionally constrained, a subset of vectors from the computed set of ADCs is accordingly selected.
- Set these ADCs as EXDCs for that output bit and simplify the circuit under this condition.
- Update the QCC with the latest available approximate circuit and iterate this process over all output bits.

The above procedure ensures that the approximations carried out never violate the specified error bounds. Also, the intermediate circuit produced after each iteration is legal and synthesis can be stopped at any point to yield a valid approximate circuit. As shown in Section 4, these steps can be realized using conventional logic synthesis tools, which vastly increases the scope of optimization techniques used in ALS.

2.3.2 SALSA Methodology

This section describes the methodology that we use to realize the approach proposed in the previous section. The potential challenges in such an implementation and the speedup techniques and heuristics to overcome the same are also described.

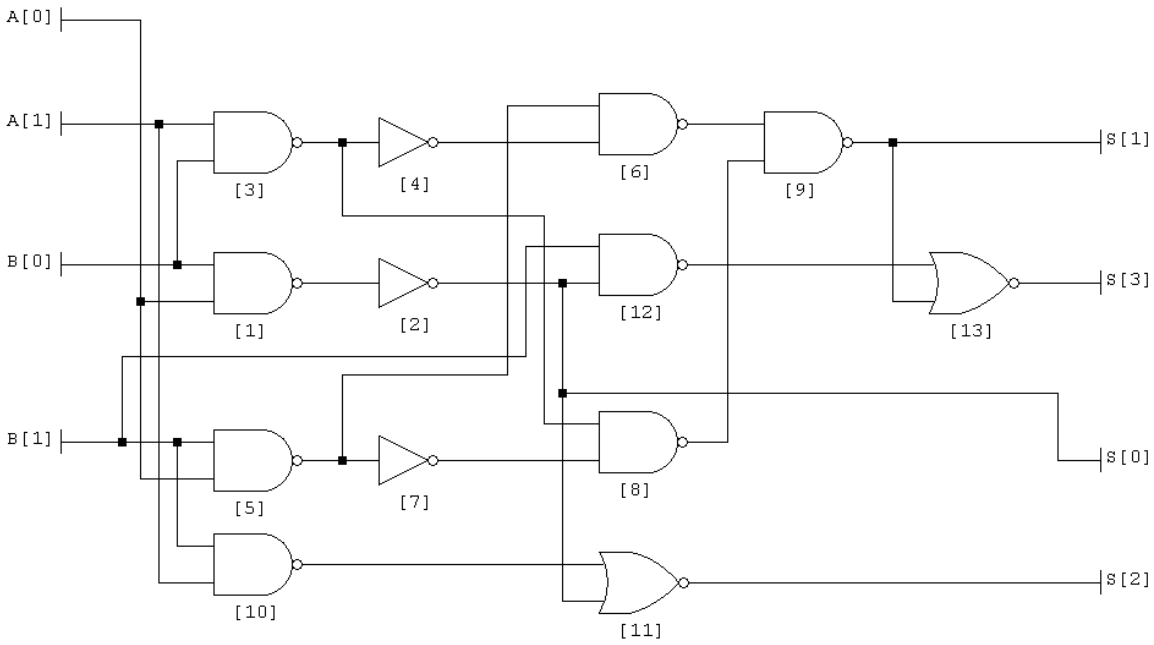


Fig. 2.6.: Illustration: 2-bit multiplier circuit

To illustrate the various steps in the methodology, we will use a multiplier circuit and the Q-function (relative error) circuit shown in Figures 2.6 and 2.7 respectively. The primary inputs to the multiplier 2-bit signals 'A','B' and the 4-bit original and approximate outputs are denoted as 'So' and 'Sa' in the Q-function.

Algorithm 1 provides an overview of the steps involved in SALSA. For each output bit, SALSA computes the ADCs and uses them to approximate the logic cone that generates it. The process of finding the ADCs for a given output bit is carried out in two steps. First, the ADCs are computed as a function of other inputs to the Q-function in the QCC *i.e.* PO_{orig} and PO_{approx} with the exception of the output bit being processed. Next, using the original and the approximate circuits, these ADCs

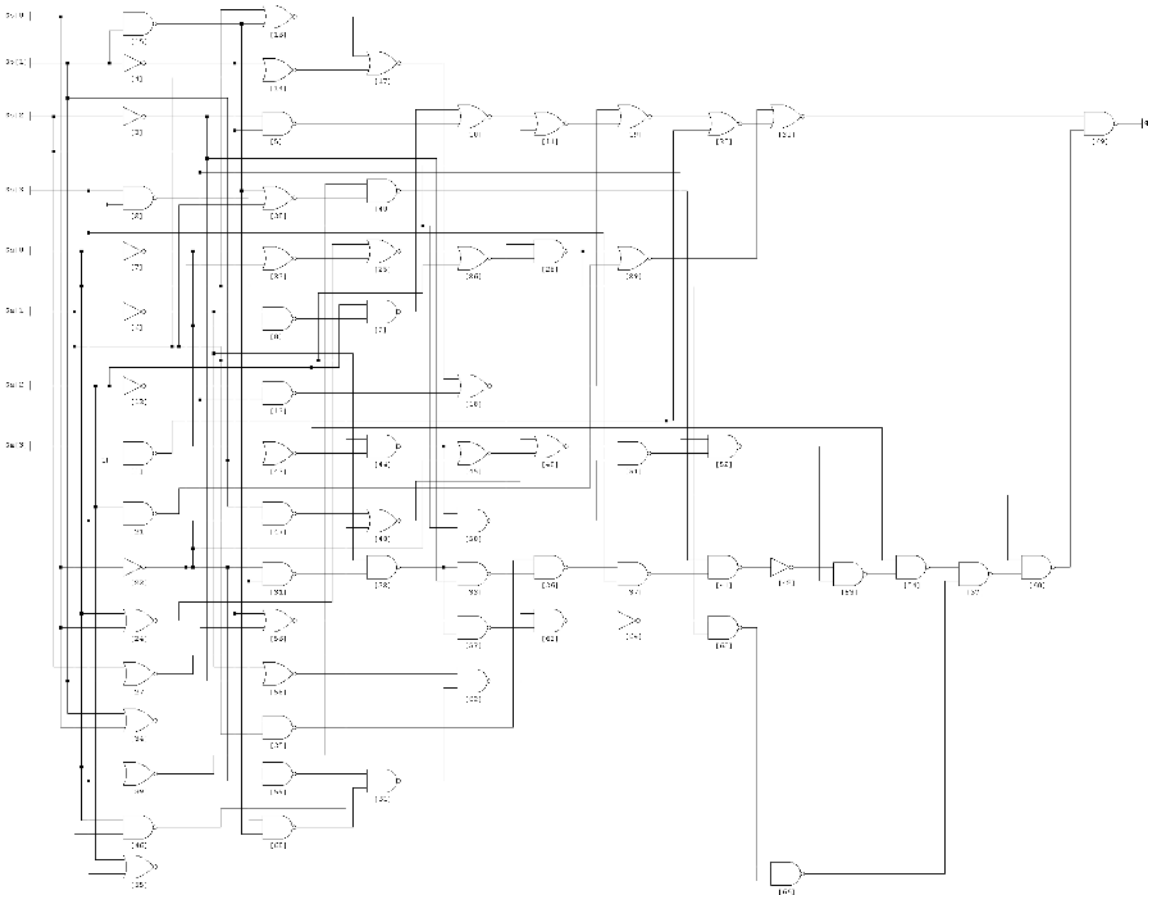


Fig. 2.7.: Illustration: Quality function

are expressed in terms of the primary inputs. After this, the computed ADCs are specified as EXDCs for the output bit under consideration and used to simplify its logic. The approximate circuit thus obtained is retained as the starting point for subsequent iterations. We describe below how these steps can be implemented using off-the-shelf logic synthesis tools.

SALSA algorithm

STEP 1: In order to compute the ADCs of a primary output bit PO_i of the approximate circuit, we should perform ODC analysis at that node in the QCC. Finding ODCs of an internal node in a circuit, as shown in Figure 2.8, involves co-

factoring the output with respect to the internal node and finding the set of input combinations for which both the positive and negative co-factors are equal. The resultant circuit contains a description of the ADCs of PO_i in terms of all outputs in PO_{orig} and all outputs in PO_{approx} except PO_i. We call this the ADC-PO circuit.

Algorithm 1 SALSA

Inputs: O : Original Circuit

Q : Quality Function

Output : A : Approximate Circuit

Begin

Initialize $A \Leftarrow O$

for each $PO_i \epsilon PO_{approx}$ **do**

STEP 1: Obtain ADCs as f($PO_{orig} \cup PO_{approx}$ - PO_i)

$ADC\_PO_i \Leftarrow$ Get_ADC_PO (Q, PO_i)

STEP 2: Obtain ADCs as f(PI)

$ADC\_PI_i \Leftarrow$ Get_ADC (O, A, ADC_PO_i)

STEP 3: Select high effort ADCs

$ADC_i \Leftarrow$ Sel_ADC (ADC_PI_i, A, ADC_i)

STEP 4: Approximate PO_i using ADCs

$A_i \Leftarrow$ Approx_PO (A, PO_i, ADC_i)

Update $A \Leftarrow A_i$

end for

Return A

End

In this step, we have essentially extracted the information about the sensitivity of Q to the primary output of interest. This step is performed only with the Q-function and does not involve the original circuit in any way. Once we have extracted this information, the Q-function is not required any further in the algorithm. For the Q-function shown in Figure 2.7, when approximating output bit Sa[1], the ADC-PO

circuit looks as shown in Figure 2.9. Note that the ADC-PO circuit is independent of Sa[1] and is a function of all original and other approximate output bits.

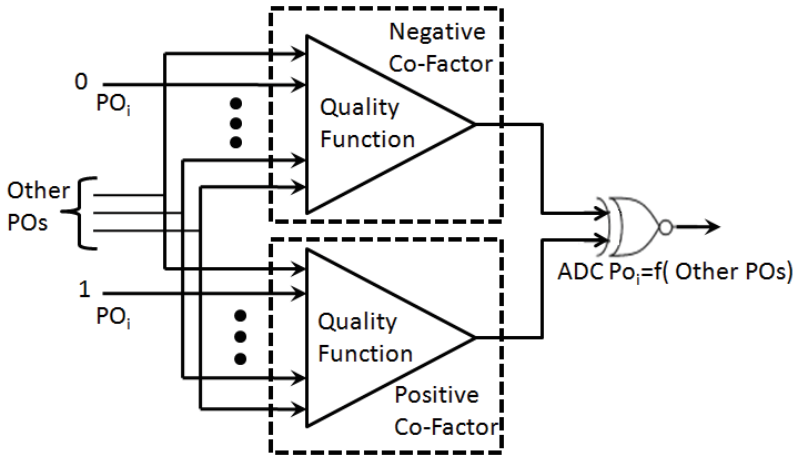


Fig. 2.8.: STEP1 - Obtaining ADCs of a primary output in terms of other original and approximate circuit outputs

STEP 2: After STEP 1, the ADCs for PO_i are available as a function of other primary outputs in the ADC-PO circuit. In this step, we express the ADCs in terms of primary inputs of the circuit. As shown in Figure 2.10, we connect the approximate and original circuits to the ADC-PO circuit obtained in the previous step. This concatenated circuit is simplified and the required ADCs for PO_i are thus obtained.

For our example, assume that the other output bits have not yet been approximated. In that case, the original circuit is same as the approximate circuit. By concatenating it to the ADC-PO circuit shown in Figure 2.9, the ADCs of S[1] in terms of primary inputs are obtained as shown in Figure 2.11.

STEP 3: Once the ADCs are computed in terms of primary inputs, if a bound on error probability is additionally provided, the high effort ADCs need to be selected for circuit approximation. If the number of ADCs are less than the target error probability, than the entire set is used. If not, the circuit is simulated for the ADC set and the value of at the internal nodes are recorded. The effort value for the ADCs are then calculated based on the number of wires that are set to non-controlling value.

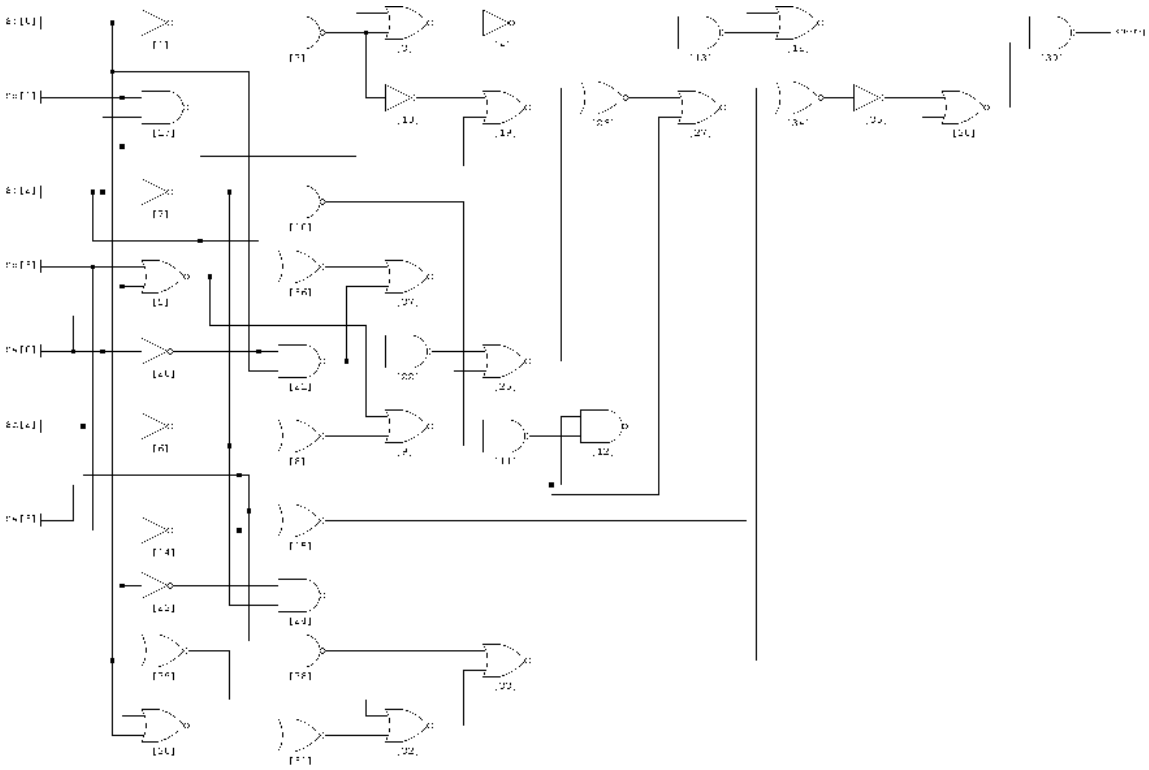


Fig. 2.9.: Illustration: ADC-PO circuit when approximating output bit S[1]

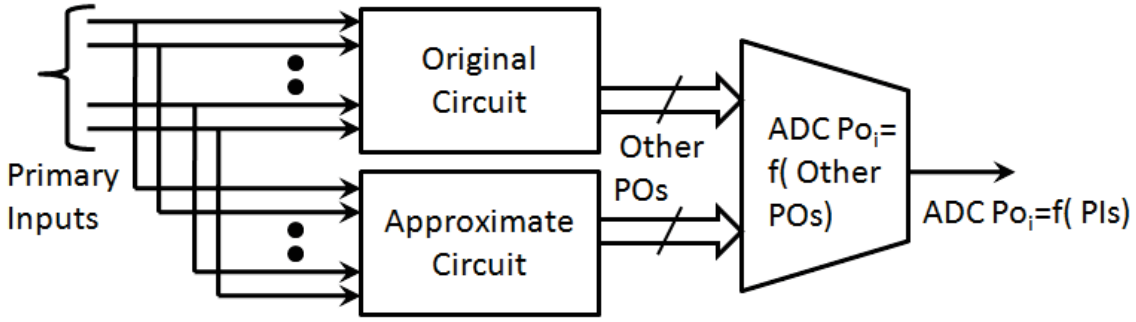


Fig. 2.10.: STEP2 - Obtaining ADCs of a primary output in terms of primary inputs

The ADCs are sorted based on the effort and the high effort ADCs are selected such that the error probability is satisfied. If the number of ADCs are large, then an exhaustive simulation is infeasible. For example, in case of error probability metric, the ADC set is comprised by the entire input space. In such a scenario, we find effort of ADC cubes rather than individual ADCs. The size of the cube is decided based

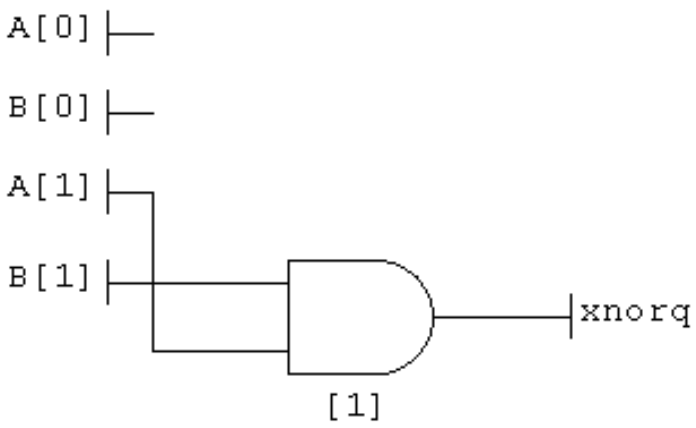


Fig. 2.11.: Illustration: ADC circuit for output S[1]

on the number of primary inputs in the circuit. By doing this we lose optimality in computing the exact ADC set but it largely contributes to scalability of the algorithm.

STEP 4: Given a set of ADCs for an output bit, approximating its logic can be done in a fairly straight forward manner. The computed ADCs are specified as External Don't Cares in the appropriate format required by the logic synthesis tool and conventional don't care based optimization techniques are invoked to simplify the logic cone that generates the output bit. The resultant circuit is used as the approximate circuit in the next iteration.

Using the ADCs obtained in Figure 2.11 , the output bit S[1] is simplified and the corresponding approximate circuit is shown in Figure 2.12. This circuit should now be used in STEP 2 to calculate the ADCs of the successive output bit.

Thus, SALSA efficiently implements the approach described in Section 3 by reformulating the ALS problem using traditional logic synthesis operations. In each iteration of the algorithm, the synthesis tool is called thrice — once to perform each of the three steps in the algorithm.

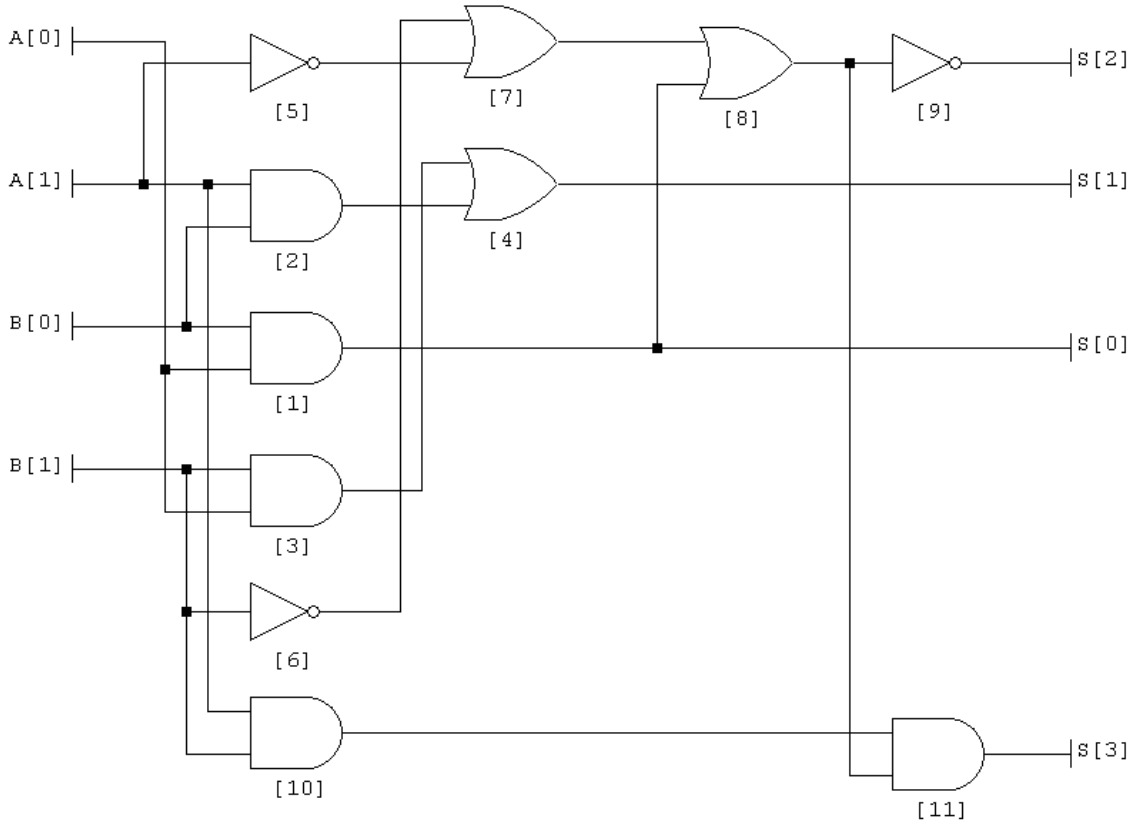


Fig. 2.12.: Illustration: Approximate circuit with output S[1] simplified

2.3.3 Speedup techniques and other heuristics

We next describe some optimizations that could be used to enhance the scalability of the SALSA methodology. The challenges to the above methodology could stem from two different sources - the quality function and the original circuit. If the Q-function is complex, the run-times of steps 1 and 2 of the algorithm are impacted. Also, if the original circuit has a large number of inputs or outputs, then forming the ADCs in step 2 could be a time consuming process. Speed up techniques and heuristics to overcome these challenges are discussed below.

Equating un-approximated output bits

In SALSA, each iteration of the algorithm approximates the logic cone that generates one output bit. The hitherto unprocessed output bits should have their logic to be same as the original circuit. Therefore, while calculating the ADCs, we need not specify the entire approximate circuit, but only the logic cones that generate the output bits that have been previously approximated.

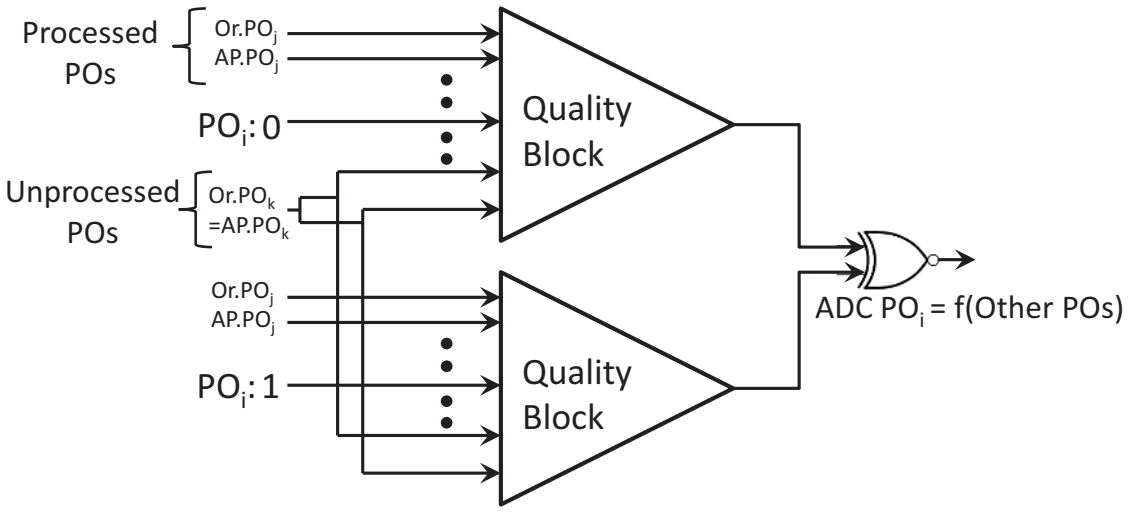


Fig. 2.13.: Equating un-approximated output bits

Using this observation, as shown in Figure 2.13, the unprocessed output bits of the approximate circuit are set equal to the corresponding output bits of the original circuit. This vastly simplifies the logic for ADC generation (STEP 1 and STEP 2), especially for initial output bits processed, and does not result in loss of any optimality in the approximations. In our example, since S[1] is the first bit to be approximated, all output bits in the approximate circuit are same as the original circuit. Hence the ADC-PO circuit in Figure 2.9 can be simplified to Figure 2.14. Notice that Figure 2.14 is only a function of original outputs. We obtain the same ADC set as in Figure 2.11 using this ADC-PO circuit.

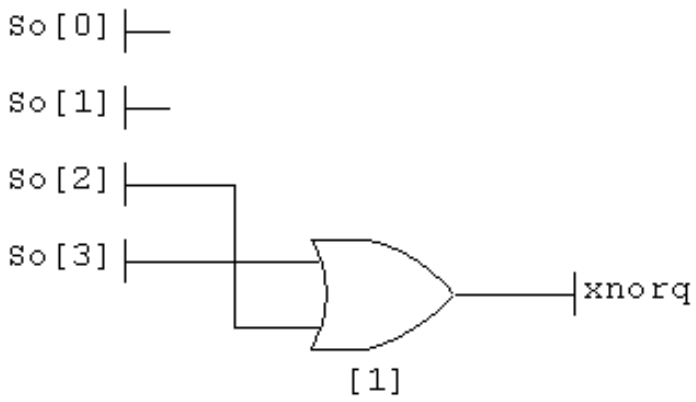


Fig. 2.14.: Illustration: ADC-PO circuit with un-approximated output bits equated

Quality function decomposition

When the number of outputs present in the original circuit is large, the complexity of the Q-function eventually grows and STEP 1 and STEP 2 of the algorithm consume significant time. A divide-and-conquer heuristic, shown in Figure 2.15, could be used to tackle this bottleneck. The idea is to decompose the Q-function into stages, with each stage only considering a disjoint subset of outputs from the original circuit. Each stage thus approximates only its subset of output bits and its output is used as the initial approximate circuit in the next stage. For the first stage, the functionality of the Q-function does not change because none of the output bits have been approximated. However, to preserve quality at the end of each stage, the subsequent stages are designed considering the worst case error that could occur in the previously approximated output bits.

The method used to decompose the Q-functions used in this work are presented below. First, consider the error magnitude metric for a circuit with N output bits. The Q-function is given by Equation 2.9.

$$Q = \big(|PO_{orig}[N-1:0] - PO_{approx}[N-1:0]| \leq K \big) \quad ? \quad 1:0 \tag{2.9}$$

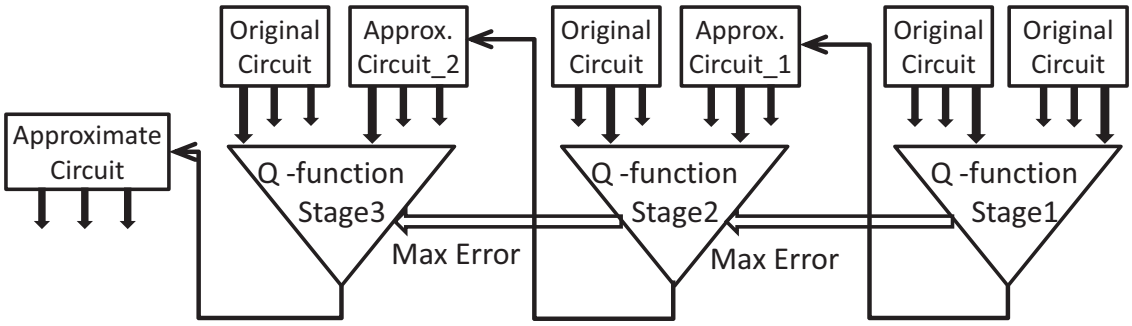


Fig. 2.15.: Quality function decomposition

The aim is to decompose this Q-function into two disjoint stages that use the lower and upper N/2 output bits respectively. The first stage is functionally similar to Equation 2.9 but uses only the lower N/2 output bits as shown in Equation 2.10.

$$Q = \big(|PO_{orig}[N/2-1:0] - PO_{approx}[N/2-1:0]| \leq K \big) \quad ? \quad 1:0 \tag{2.10}$$

The first N/2 bits can be approximated using this as the Q-function. Now for approximating the next N/2 bits, the second stage is constructed by conservatively assuming the maximum error that could occur in the lower N/2 bits. The Q-function for the second stage is given by Equation 2.11.

$$Q = \big(|PO_{orig}[N-1:N/2] - PO_{approx}[N-1:N/2]| \leq K - 2^{N/2} - 1 \big) \quad ? \quad 1:0 \tag{2.11}$$

This procedure can be extended to further break the Q-function into as many stages as desired.

The relative error metric given in Equation 2.2 could similarly be decomposed into disjoint stages. For an N-bit circuit, to decompose the Q-function into K stages, the i^{th} stage will be given together by Equations 2.12a, 2.12b and 2.12c.

$$Q1 = \left(\frac{PO_{approx}[\frac{Ni}{K} - 1 : \frac{N(i-1)}{K}] + 1}{PO_{orig}[\frac{Ni}{K} - 1 : \frac{N(i-1)}{K}]} \leq 1 + K \right) \quad ? \quad 1:0 \tag{2.12a}$$

$$Q2 = \left(\frac{PO_{approx}[\frac{Ni}{K} - 1 : \frac{N(i-1)}{K}]}{PO_{orig}[\frac{Ni}{K} - 1 : \frac{N(i-1)}{K}] + 1} \geq 1 - K \right) \quad ? \quad 1:0 \tag{2.12b}$$

$$Q = Q1 \quad \& \quad Q2 \tag{2.12c}$$

Equation 2.12a satisfies the upper bound of Equation 2.2 by propagating its worst case when the value of numerator is maximized because of errors in previous stages and the denominator remains unaffected by any error. Equation 2.12b propagates worst case for the lower bound of Equation 2.2 which occurs when the denominator value is maximized while the the numerator remains unaffected. The Q-bit is the logical *AND* of both lower and upper bounds as given in Equation 2.12c.

This divide and conquer heuristic is very powerful because it allows SALSA to handle Q-functions of arbitrary sizes by suitably decomposing them into disjoint stages. Nevertheless, we sacrifice some approximation capability in using this heuristic as the maximum error is propagated across Q-blocks in adjacent stages.

Exploiting input-output dependencies

The previous speedup techniques were targeted at addressing the challenge of the Q-function being complex or having a large number of inputs. However, challenges may arise in finding the ADCs when the circuit to be approximated itself is large. We present two techniques to directly address this issue.

We know that, for a given output bit, not all primary inputs lie in its cone of logic. Hence, when finding the ADCs (in STEP 2) for a given output bit, we just need to define the circuit in terms of primary inputs in its transitive fan-in and generate the ADCs only in terms of these inputs. This is accomplished in Figure 2.16 (dotted lines), where the independent primary inputs are removed from the original and approximate circuit blocks. It is important to note that, this technique is exploited when generating the ADCs (STEP 2) and not when simplifying the circuit using these ADCs (STEP 3). This is because we would like to preserve the logic sharing between the output bits. In the implementation, we use the IO dependencies for ADC generation but do not extract cones of logic during logic simplification.

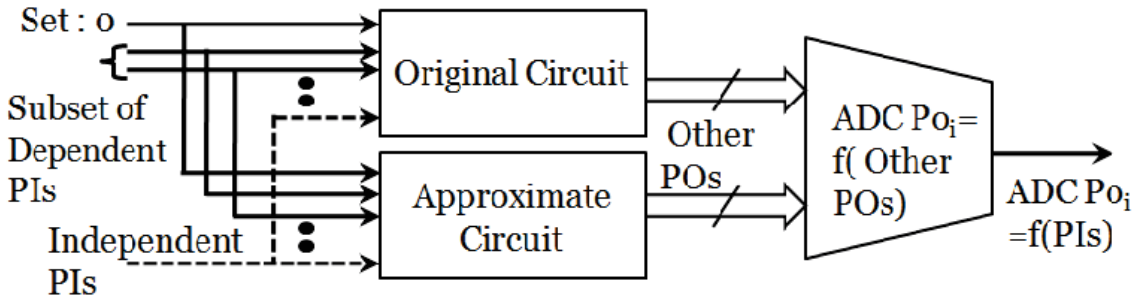


Fig. 2.16.: Exploiting I/O dependencies and calculating subset of ADCs

Calculating subset of ADCs

Although the above technique is efficient in many cases, it is not effective when a primary output depends on most of the primary inputs. This scenario happens in the output MSB bits of arithmetic circuits, where the output depends on all less significant input bits. To tackle this, we resort to computing only a subset of the ADCs and use them for circuit approximation. As shown in Figure 2.16, in STEP2, we set certain dependent inputs in the cone of logic of an output to zero and then calculate its ADCs using the usual procedure. In the calculated ADC set, the condition for the dependent inputs that were set to zero is appropriately added before using them for circuit approximation. The above techniques allow us to use SALSA on larger circuits and more complex Q-functions.

2.3.4 Experimental Methodology

In order to demonstrate the proposed approach, we tested it on a wide range of circuits for three different quality metrics descirbed in Section 2.2 *viz.* maximum error magnitude (Equation 2.1), relative error (Equation 2.2) and error probability (Equation 2.5). Further, to demonstrate generality, the methodology was implemented using two different off-the-shelf synthesis tools, namely SIS [33] and Synopsys Design Compiler [34].

Table 2.1.: Circuits used in experiments to evaluate SALSA

Name	Delay (ns)	*P_orig* Width	Gate Count	I/O
RCA	Ripple Carry Adder	32	1012	64/33
KSA	Kogge Stone Adder	32	1361	64/33
CLA	Carry Look-ahead Adder	32	926	64/33
MUL	Array Multiplier	8	1055	16/16
WTM	Wallace Tree Multiplier	8	1132	16/16
MAC	Multiply and Accumulate with 32-bit accumulator	8	1910	48/33
SAD	Sum of Absolute Differences (Used in Motion Estimation)	8	1241	48/33
EU_DIST	2D-Euclidean Distance Unit (sans square root)	8	1668	32/16
BUT	Butterfly structure (Used in FFT computation)	8	496	16/18
FIR	4-tap FIR filter	8	1719	32/16
IIR	4-tap IIR filter	8	2135	56/16
DCT	8-input Discrete Cosine Transform Block	8	10817	64/72

The circuits used in the experiments, listed in Table 2.1, range from simple arithmetic circuits to complex datapaths. The complexity of the circuits in terms of number of inputs, outputs, and gate count is also listed. The circuits were mapped to the IBM 45nm technology library using Design Compiler and evaluated for area, power and delay using Synopsys Power Compiler.

2.3.5 Results

In this section, we present the results of experiments that evaluate the approximate circuits generated by SALSA. We begin by presenting the area and power savings obtained at iso-delay for the benchmarks listed in Table 2.1 across various error metrics. We demonstrate the effectiveness of SALSA approximations across all design points of the circuit by comparing the original and approximate circuit implementations for the entire delay range. We illustrate the scalability of SALSA by synthesizing circuits of various bit widths and show that the approximations scale across the same. We also perform a detailed quality function exploration to understand the nature of approximations carried out by SALSA. Finally, we show for error significance metric that SALSA approximated circuits outperform circuits approximated by output LSB truncation for all target error values.

Area and power benefits for various error metrics

Maximum error magnitude

Figure 2.17 shows the relative area (ratio of approximate circuit to original circuit) *vs.* maximum error magnitude and relative power (ratio of approximate to original) *vs.* maximum error magnitude plots for the benchmark circuits. The maximum error magnitude is shown as a percentage of the maximum output value because the circuits possess different numbers of output bits and thus errors of the same magnitude have varying significance. The dynamic ranges of feasible errors are accordingly different, prompting the use of 2 different error ranges in the graphs. For 32-bit circuits like adders, MAC, and SAD, the lower X axis scale is used while other circuits, whose individual outputs have fewer bits (9 for DCT, BUT and 16 for the rest), follow the upper X axis. From the results, we see an exponential decrease (Note: X axis is in log scale) in area and power initially, which then commences to taper out as we move towards larger error values. This is explained by the fact that, in any arithmetic

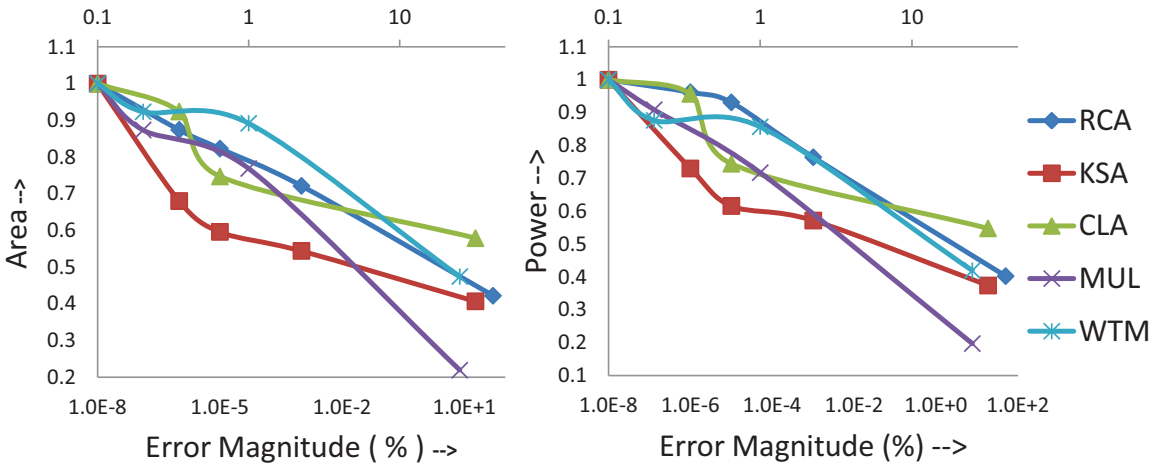
circuit, adjacent output bits have an exponential difference in their significance. So, for the same increase in maximum error magnitude, the incremental potential for approximation is less as the actual value of the error increases. Also, for a given maximum error magnitude, the cone of logic generating the LSB bits, that have exponentially lower significance compared to their MSB counterparts, have a large set of ADCs and hence have a better chance of being approximated. From Figure 2.17, we see that SALSA yields area savings in the range of 1.1X-1.85X for tight error constraints (less than 1%) and up to 4.75X for relaxed error constraints (upto 20%). Power benefits range from 1.15X-1.75X and 1.3X-5.25X for similar tight and relaxed error constraints respectively.

Relative error

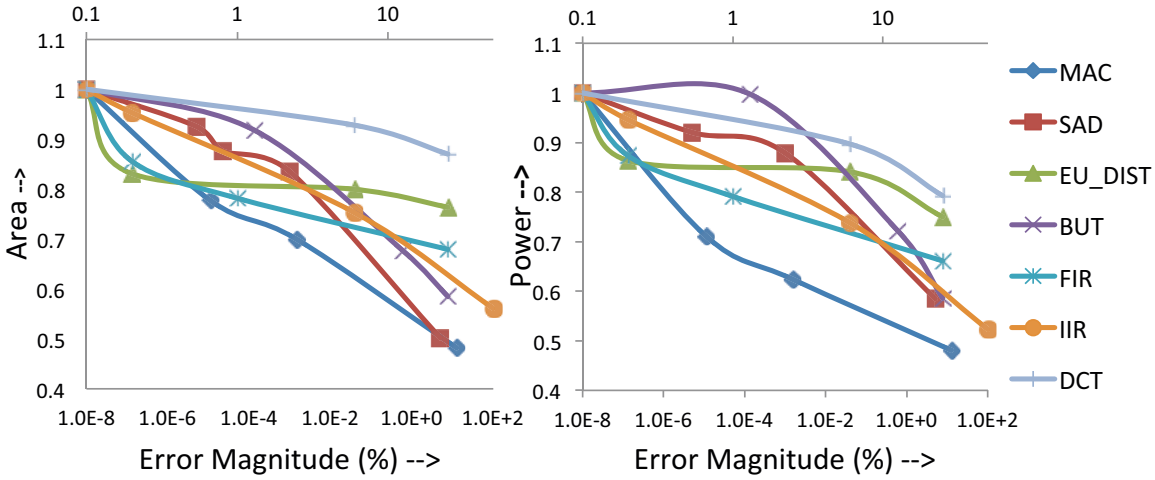
The next set of graphs, in Figure 2.18, show the results obtained for the relative error metric. The relative error is defined as the ratio of the approximate output to the original output. Similar trends with savings up to 1.7X in area and 1.65X in power are observed for this metric. We also observed that the ADCs derived by SALSA for the relative error metric and the maximum error magnitude metric differed significantly. In case of the relative error metric, for small actual values of output, even a small change in the logic would prompt the output to deviate by a large percentage relative to the golden value. Moreover, the ADCs for the LSBs cannot depend on the MSB inputs. Therefore, we get a comparatively larger ADC set for the MSB output bits.

Error probability

Graphs shown in Figure 2.19 give the area and power reduction obtained by approximating the benchmark circuits for error probability metric. As seen from Figure 2.19a, in case of arithmetic circuits, the area and power savings range upto 1.53X and 1.45X respectively for tight error probabilities upto 5% and upto 2.5X for relaxed error probabilities. Similarly, for complex data paths shown in Figure 2.19b, the sav-



(a) Area and power savings for arithmetic circuits



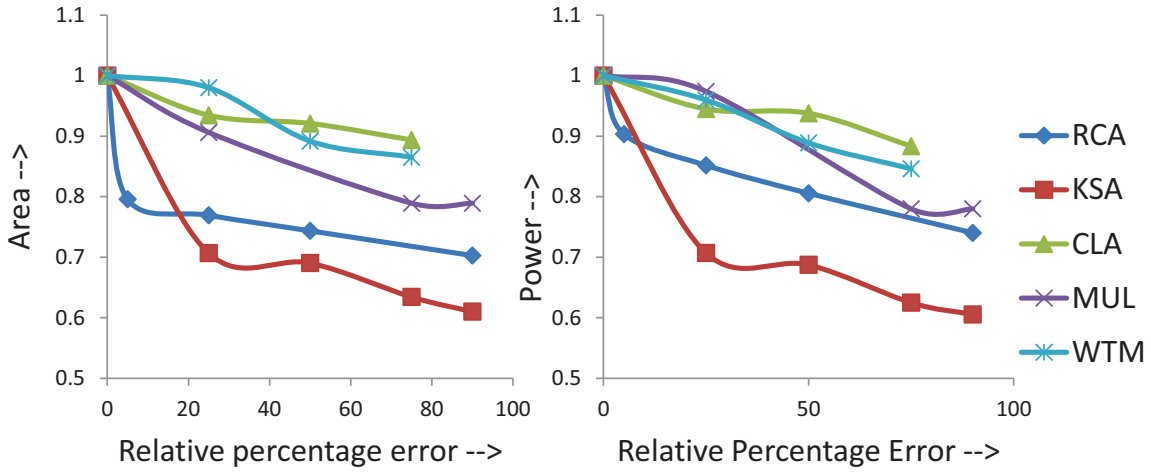
(b) Area and power savings for complex blocks and complete datapaths

Fig. 2.17.: Results for maximum error magnitude metric

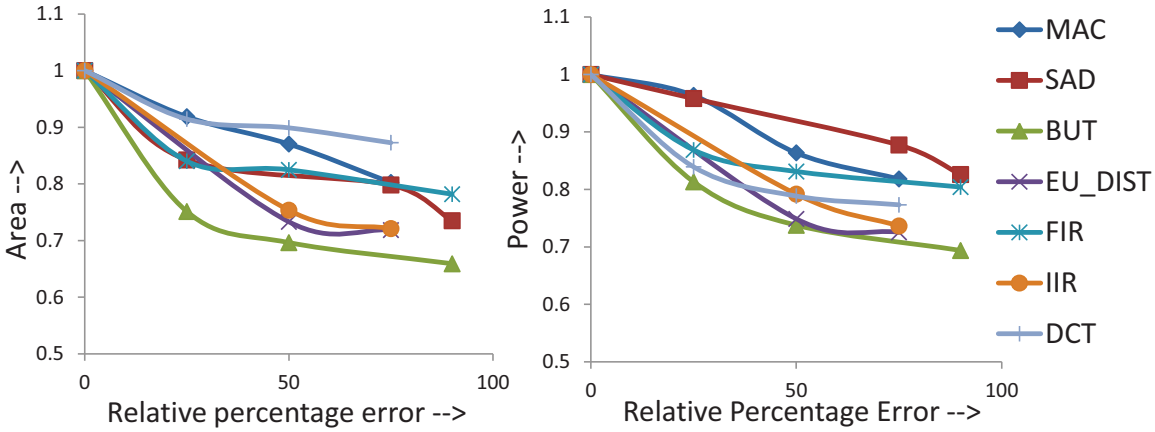
ings in area and power range upto 1.67X and 2.3X for tight and relaxed constraints respectively.

Area and power comparison across circuit delay range

Functional approximations facilitated by SALSA have the potential to improve area, power or performance or combinations thereof. Each curve in the graphs pre-



(a) Area and power savings of arithmetic circuits

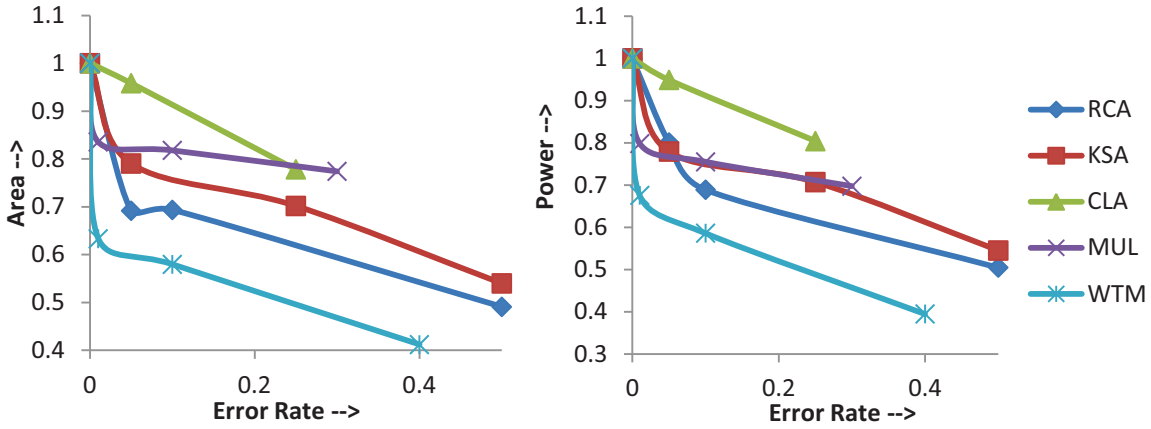


(b) Area and power savings of functional blocks and datapath modules

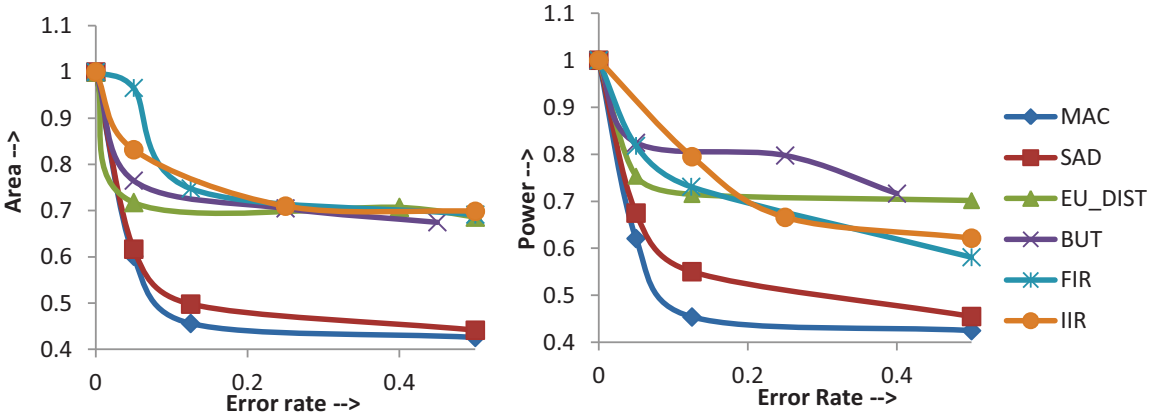
Fig. 2.18.: Results for relative error metric

sented in section 6 contains iso-delay points for a given delay of the original circuit. To conclusively demonstrate the effectiveness of the functional approximation over a wide range of delay values, we synthesize the approximate and original circuits for the entire delay range of the original circuit and plot the corresponding Area *vs.* Delay and Power *vs.* Delay curves for various error values.

Figure 2.20 shows the delay sweep plot for a 32-bit Kogge-Stone adder for the maximum error magnitude quality metric. Figure 2.20a gives the actual area and



(a) Area and power savings for arithmetic circuits

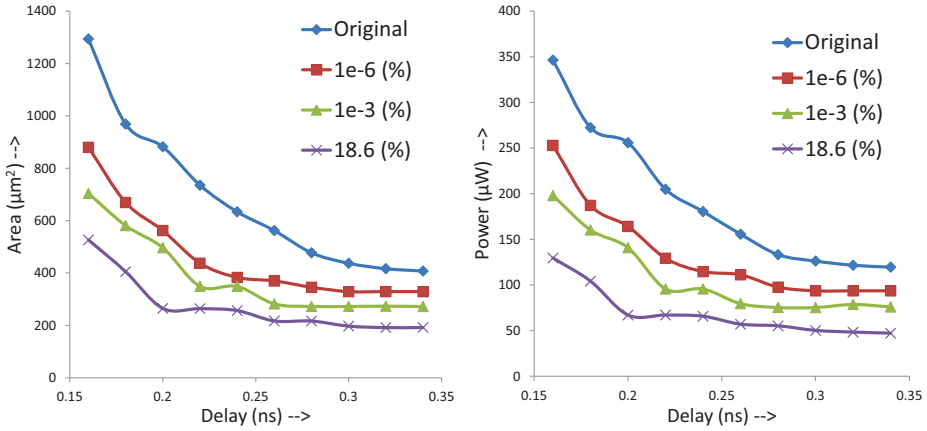


(b) Area and power savings for complex blocks and complete datapaths

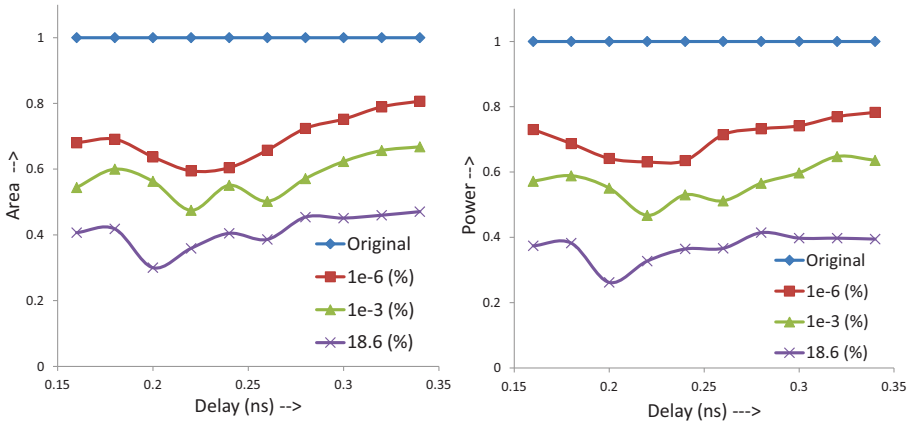
Fig. 2.19.: Results for error probability metric

power values while the normalized values are depicted in Figure 2.20b. We could see that the functionally approximated circuit performs consistently better in area and power for the entire delay range of the original circuit.

The Kogge-Stone adder is a balanced tree structure that is well optimized for delay and hence, despite considerable area and power benefits, the scope for reducing circuit delay is minimal. On the other hand, if we consider a linear structure like the ripple-carry adder which is optimal in area but not in delay, the approximations result in breaking carry chains, thereby yielding significant performance benefits. Figure 2.21



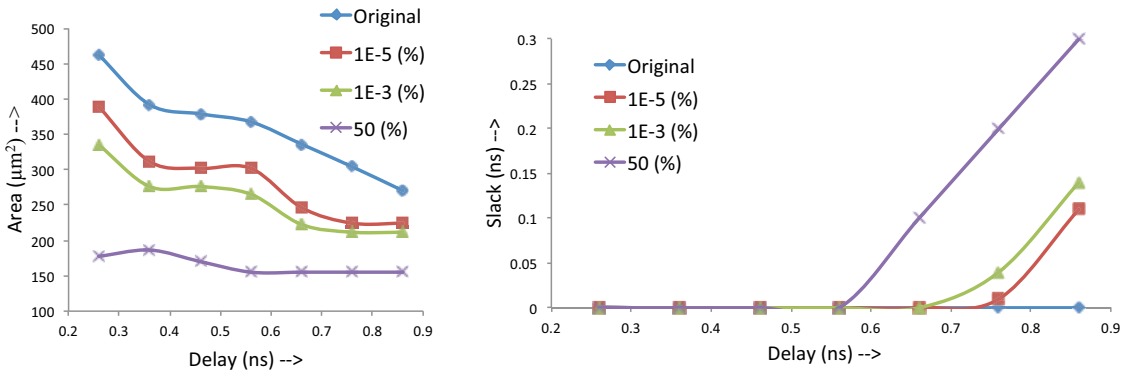
(a) Area *vs.* Delay and Power *vs.* Delay Trade-off Curves



(b) Normalized Area *vs.* Delay and Normalized Power *vs.* Delay Curves

Fig. 2.20.: Delay sweep of original and approximate circuits for a 32-bit kogge stone adder

shows the delay sweep plot for a 32-bit ripple Carry adder for the same quality metric. The slack obtained in each implementation is provided in Figure 2.21b. Although the area savings saturate after a point where structural or sizing optimizations do not yield any improvement, we see a corresponding rise in the circuit slack. The approximate circuit could thus be clocked at a higher frequency or alternatively be subject to voltage scaling (without any timing errors) to instead obtain additional savings in power. In summary, the above plots illustrate the fact that SALSA offers a wide scope of approximation across all design points for a given circuit.



(a) Area *vs.* Delay Sweep

(b) Circuit Slack *vs.* Delay Sweep

Fig. 2.21.: Delay sweep of original and approximate circuits for a 32-bit ripple carry adder

Scaling of SALSA approximations across circuit bit widths

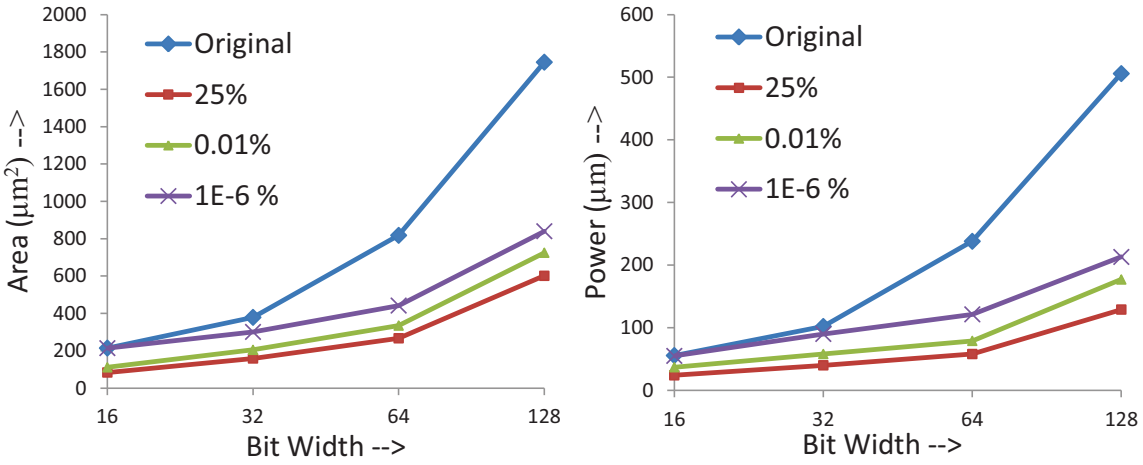


Fig. 2.22.: Area and power scaling for adders of various bit widths

The graphs shown in Figure 2.22 depict the area and power benefits obtained across adders of different bit widths (16,32,64,128) for the same maximum error magnitude. The scalability of the proposed approach is demonstrated by the fact that SALSA is able to consistently achieve benefits for similar circuits with varying bit widths.

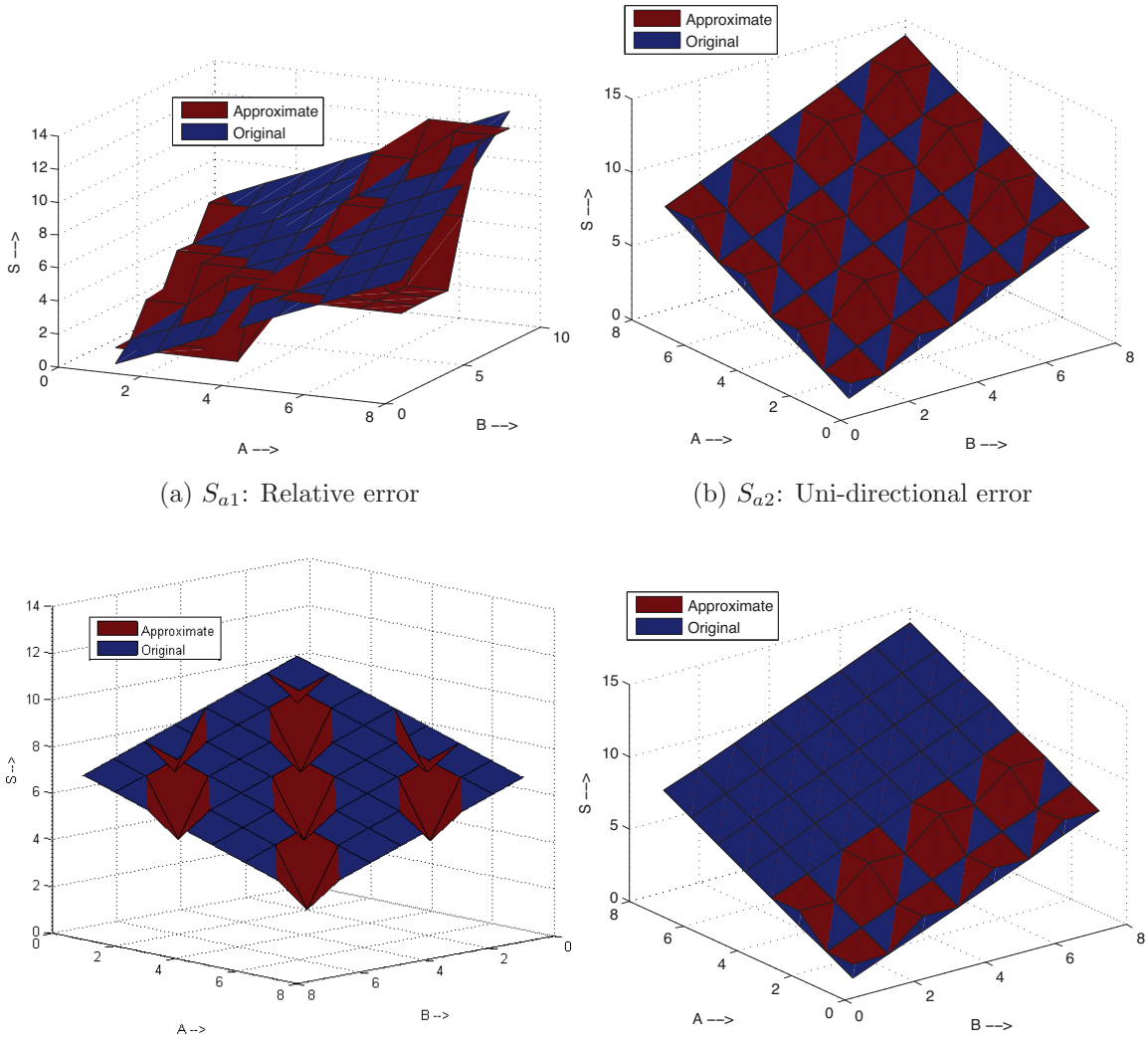
Quality function exploration

In order to provide few insights into the nature of approximations performed by SALSA, we use it to approximate a 3-bit adder and exhaustively simulate the resulting circuit for all possible input combinations. We choose 4 different error metrics *viz.* relative error, uni-directional relative error, error probability and uni-directional error with error probability. The relative error metric was described earlier, while the uni-directional percentage error metric has the additional constraint that errors may occur only in a certain direction (Positive or Negative). In this case, the quality function was coded such that the approximate circuit can tolerate errors only in the positive direction (*i.e.,* approximate result could only be greater than golden value). We choose an error bound of 75% in both cases for illustration. The error probability metric is the percentage of inputs that can produce a wrong value, which in this case is chosen to be 15%. The uni-directional error with error probability metric, in addition to allowing only positive errors, bounds the error probability to 15%. Table 2.2 shows the results of the simulation performed. Columns A and B in the table are circuit inputs, while columns S_o, S_{a1} through S_{a4} represent outputs corresponding to original circuit and approximate circuits corresponding to the various quality metrics. From the table, we can see that the error bounds are never violated in the approximate circuit. Also in the S_{a2} column of the table, corresponding to the uni-directional error metric, the approximate circuit has errors only in the positive direction. From the table, we can also observe that the outputs of the approximate circuit tend to be clustered together based on the available resilience. This clustering of outputs for adjacent input combinations enhances the scope of approximation because, from a truth table perspective, adjacent entries have the same value for all output bits and the resultant function thus has a better chance for being simplified. For the uni-directional error metric, since we can tolerate only positive errors, there is less scope of clustering when the actual output is itself large. Accordingly, we could see

Table 2.2.: Truth table comparison of original and approximate 3-bit adder for relative error (S_{a1}), uni-directional relative error (S_{a2}), error probability (S_{a3}) and unidirectional relative error with error probability (S_{a4}) metrics

A	B	S_o	S_{a1}	S_{a2}	S_{a3}	S_{a4}	A	B	S_o	S_{a1}	S_{a2}	S_{a3}	S_{a4}
0	0	0	1	1	0	1	4	0	4	5	5	4	4
0	1	1	1	1	1	1	4	1	5	5	5	5	5
0	2	2	1	3	2	3	4	2	6	5	7	6	6
0	3	3	1	3	3	3	4	3	7	5	7	7	7
0	4	4	5	5	4	5	4	4	8	9	9	8	8
0	5	5	5	5	5	5	4	5	9	9	9	9	9
0	6	6	5	7	6	7	4	6	A	9	B	A	A
0	7	7	5	7	7	7	4	7	B	9	B	B	B
1	0	1	1	1	1	1	5	0	5	5	5	5	5
1	1	2	1	2	2	2	5	1	6	5	6	6	6
1	2	3	1	3	3	3	5	2	7	5	7	7	7
1	3	4	5	4	4	4	5	3	8	5	8	8	8
1	4	5	5	5	5	5	5	4	9	9	9	9	9
1	5	6	5	6	6	6	5	5	A	9	A	A	A
1	6	7	5	7	7	7	5	6	B	9	B	B	B
1	7	8	5	8	8	8	5	7	C	D	C	C	C
2	0	2	3	3	2	3	6	0	6	7	7	6	6
2	1	3	3	3	3	3	6	1	7	7	7	7	7
2	2	4	5	5	4	5	6	2	8	5	9	8	8
2	3	5	5	5	5	5	6	3	9	5	9	9	9
2	4	6	7	7	6	7	6	4	A	B	B	A	A
2	5	7	7	7	7	7	6	5	B	B	B	B	B
2	6	8	5	9	8	9	6	6	C	D	D	C	C
2	7	9	5	9	9	9	6	7	D	D	D	D	D
3	0	3	3	3	3	3	7	0	7	7	7	7	7
3	1	4	5	4	4	4	7	1	8	5	8	8	8
3	2	5	5	5	5	5	7	2	9	5	9	9	9
3	3	6	5	6	6	6	7	3	A	5	A	A	A
3	4	7	7	7	7	7	7	4	B	B	B	B	B
3	5	8	5	8	8	8	7	5	C	D	C	C	C
3	6	9	5	9	9	9	7	6	D	D	D	D	D
3	7	A	5	A	A	A	7	7	E	D	E	E	E

minimal clustering towards the end compared to when the actual output values are small.



(a) S_{a1}: Relative error

(b) S_{a2}: Uni-directional error

(c) S_{a3}: Error probability

(d) S_{a4}: Unidirectional error with error probability

Fig. 2.23.: Surface plots of exhaustive simulation on approximate circuits synthesized using various quality metrics

To visualize the approximate outputs better, Figure 2.23 shows the surface plot of the original and approximate circuit outputs for each error metric. In Figure 2.23a, the approximate circuit output deviates from the original in both directions while the unidirectional error (Figure 2.23b) restricts the approximate output to be always above the original. Note that the case when original output is zero is a don't care

in the Q-function (divide by zero) and we see that SALSA has considered this and set the approximate output to '1' because it leads to an optimized implementation. The error probability can be visualized from the percentage of surface area where the original and approximate output planes are different. The number of spikes in Figure 2.23c corresponds to the circuit error probability which is 12.5%. Figure 2.23b, corresponding to S_{a2}, has 16 spikes and hence an error probability of 25%. But for S_{a4} this needs to be bounded to 15%. From the ADC set computed for S_{a2}, the high effort ADCs are selected and used. This bounds the error probability to the desired value as seen in Figure 2.23d. A variety of quality functions tailored suitably to the needs of application can thus be used in SALSA and the procedure for synthesis, while being independent of the error metric, efficiently exploits this flexibility in approximating the circuit.

Comparison with output LSB truncation

One direct way of constructing approximate circuits for maximum error magnitude metric is output bit truncation, in which the LSB bits of the output that carry less significance than the target error threshold can be pruned without violating quality constraints. Figures 2.24 and 2.25 show the area and power benefits obtained by

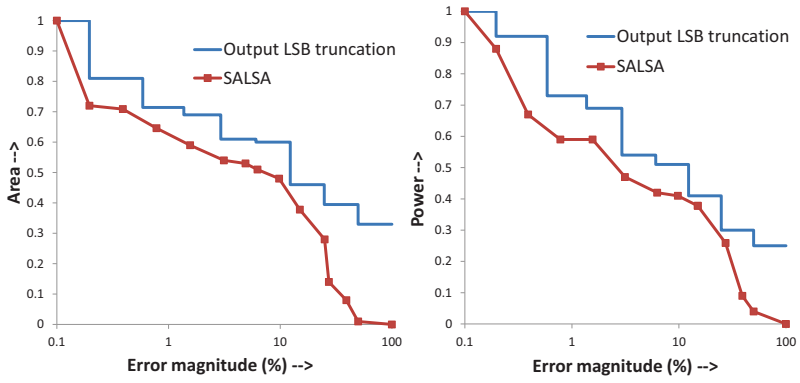


Fig. 2.24.: Area and power comparison of SALSA with output LSB truncation for kogge-stone Adder

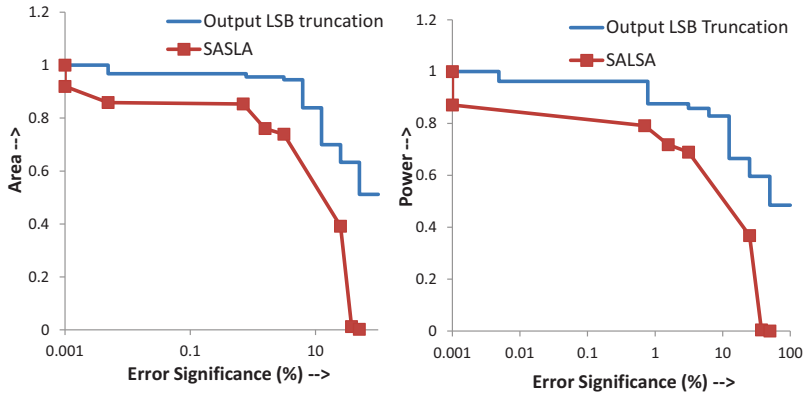


Fig. 2.25.: Area and power comparison of SALSA with output LSB truncation for array multiplier

output LSB truncation for a kogge-stone adder and array multiplier circuit. We see in case of output LSB truncation that as the error constraints are relaxed the resultant increase in savings are discretized by the number of output bits present in the circuit. The discrete steps grow exponentially in length (Note: X axis is in log scale) corresponding to the difference in significance of adjacent output bits and no additional approximation can be done for any intermediate values of errors. The graphs also depict the savings obtained by SALSA for a range of error significance constraints. We could observe that the circuits synthesized by SALSA are superior compared to output LSB truncation. This proves the ability of SALSA to make use of the flexibility offered by the error constraints during synthesis. It is worth mentioning that output LSB truncation is not possible for other metrics like relative error and error probability, while SALSA, in addition to being better, is a more general procedure for approximate logic synthesis.

2.4 SASIMI: A Unified Circuit Transformation for Approximate and Quality Configurable Circuit Design

This section describes the second approach, called **S**ubstitute-**A**nd-**Sim**pli**f**y (SASIMI), which is a unified circuit transformation for the design of both approximate and quality configurable circuits. The key insight behind SASIMI is to identify near-identical signal pairs, or signal pairs that assume the same value with high probability, and substitute one for the other. Such substitutions may introduce functional approximations — when an input causes the substituted signal to assume an incorrect value, and also causes the incorrect value to propagate to a circuit output. Naturally, it is important to restrict the substitutions chosen such that the impact on output quality is not excessive. On the other hand, well-chosen substitutions can lead to the simplification of the circuit by eliminating some of the logic that generates the substituted signal, while also downsizing logic in its transitive fan-out (since the substitution may introduce timing slack). Approximate circuits are automatically synthesized by iterating the substitute-and-simplify steps in a quality-constrained loop.

We extend SASIMI to the synthesis of quality configurable circuits, by augmenting the approximate circuit to detect when errors are incurred at runtime, and utilizing an additional clock cycle to re-compute the logic in the transitive fanout of the substituted signal, thereby "correcting" the error. The error correction may be performed universally or selectively, or completely skipped based on the desired accuracy level. While the accurate mode of operation is reminiscent of variable-latency circuits [35–38], we note that our design and synthesis approach differ significantly as the quality constraints imposed in the other approximate modes are considered. In contrast, traditional variable-latency design methodologies are oblivious to the degradation in output quality, since errors are always corrected.

In summary, the key contributions of SASIMI are as follows:

- SASIMI introduces a novel approximate circuit transformation, Substitute-and-Simplify, that judiciously employs signal substitutions to reduce the circuit's power consumption while satisfying the user-specified quality constraints.

- It is the first approach to automatically synthesize quality configurable circuits. SASIMI achieves quality configurable execution by selectively utilizing an additional clock cycle to correct errors that may violate the desired quality.

SASIMI is prototyped and evaluated across a wide range of arithmetic units, data paths and ISCAS85 benchmarks, demonstrating significant benefits in power and area. Further, case studies of using the circuits generated by SASIMI in two recognition applications — Support vector machines (SVM) and K-Nearest Neighbor (k-NN) classification – illustrate the utility of the proposed techniques.

2.4.1 SASIMI: Design Approach

Figure 2.26 illustrates the basic approach adopted in SASIMI. The key idea is to identify pairs of signals in the circuit that are similar to each other in their logic functionality and substitute one signal in place of the other, thereby functionally approximating the circuit. The signal that is being replaced is called the target signal (TS) and the signal substituting TS is termed the substitute signal (SS). The candidates for SS can be logic zero, logic one and other signals (and their complements) in the circuit. If chosen judiciously, substitutions have the ability to bring about circuit simplifications (detailed below) that yield significant savings in area and power. For approximate circuit design, the substitute-and-simplify steps are performed successively until the approximate circuit reaches the target error constraint.

The trade-offs and desired criteria in choosing the TS-SS substitution pair are shown in Figure 2.27. When the target signal gets substituted, the gates that are exclusive to the cone of logic that generates TS can be deleted. The logic in the transitive fan-out of TS, whose timing requirements are constrained by TS, is potentially downsized since the substitution with SS introduces timing slack. Further, the

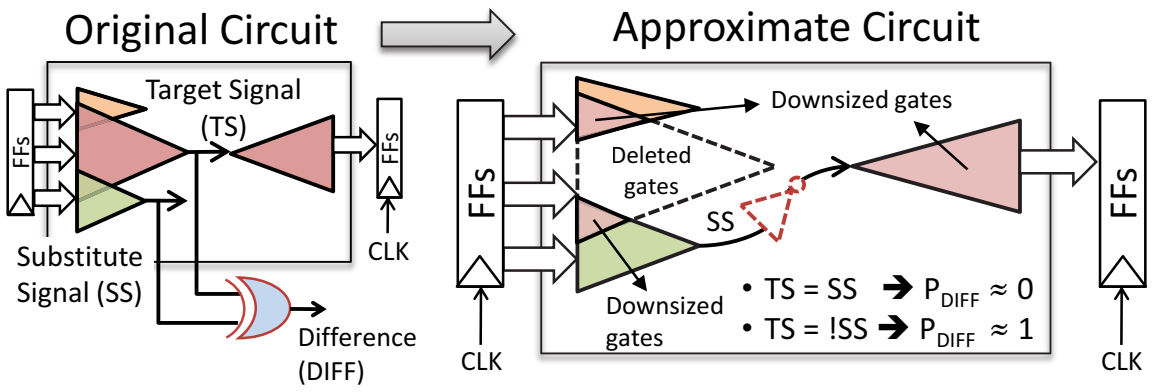


Fig. 2.26.: Approximate circuit design using SASIMI

transitive fan-ins of TS that fan-out to other logic cones in the circuit can be sized independent of TS. Thus, in choosing TS, both the direct effect of logic elimination and the indirect impact of logic downsizing should be considered. Also, signals that fan-out to outputs that cannot tolerate errors, or signals that cause unacceptable degradation in output quality, are undesirable choices for TS.

Fig. 2.27.: Criteria for selecting substitution candidates

In choosing the substitute signal, the potential error caused by the substitution should be considered. The error introduced can be inferred using the signal probability of the difference signal (P_{DIFF}), which is the XOR of TS and SS. Since each signal

and its complement are SS candidates, the ones with least probability product - $P_{DIFF} * (1 - P_{DIFF})$ - of being different from TS is desired. However, it is worth noting that an error at TS need not be sensitized at the primary outputs and hence the actual circuit error has to be separately estimated. Further, to facilitate logic downsizing, substitute signals that introduce as larger slack at TS are desirable. We also impose a constraint that substitutions should not cause combinational cycles in the circuit.

2.4.2 Quality Configurable circuit design using SASIMI

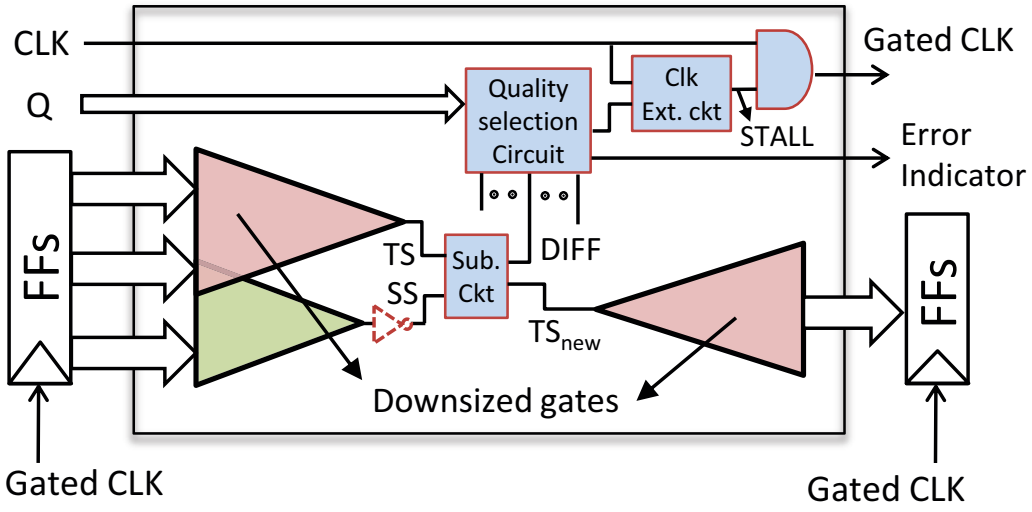


Fig. 2.28.: Quality configurable circuit design using SASIMI

The above approach can be directly applied to the synthesis of quality configurable circuits with multiple quality modes. Without loss of generality, the approach is explained considering two quality modes - the accurate mode and the approximate mode. The key idea is to make the quality configurable circuit latency elastic and "recover" from errors caused due to approximations when the circuit is in the accurate mode. As illustrated in Figure 2.28, substitutions are performed in the circuit but the logic generating TS is retained and the difference between TS and SS is monitored.

In the accurate mode, the circuit operates in a single cycle if TS and SS take the same value. Otherwise, an additional cycle is provided in which the correct result is re-computed from the point of substitution. In the approximate mode, since the error caused by the substitution is tolerable, any difference between TS and SS is ignored and the circuit always operates in a single cycle. Thus, based on the quality mode, the circuit selectively recovers from errors caused by substitutions that are intolerable. As shown in Figure 2.28, additional control inputs Q to the circuit indicate the desired quality mode.

Realizing quality configurable operation requires additional circuits for selective substitution, quality selection and clock extension, which are shown in Figure 2.29. The corresponding timing diagram illustrating single cycle and two cycle operations is given in Figure 2.30. The substitution circuit detects the difference between TS and SS, and allows the clock extension circuit to choose which of these signals is fed to downstream logic. At the start of every operation, the SS_i signals are chosen by default and the difference ($DIFF_i$) from all substitutions are accumulated (DS_{acc}). The quality selection circuit uses the accumulated difference signal (DS_{acc}) and the quality control bits (Q) to determine the need for a second clock cycle. This is used along with input quality indicator (Q) bits (Figure 2.30: Cycles 2 and 4) in the quality selection logic to determine the need for a second clock cycle. If required, the clock extension circuit then gates the clock for a cycle and (Figure 2.30: Cycle 2) also sets the EFF flip-flop so that all substitution circuits select the corresponding TS_i signals in the second cycle, thereby correcting the error.

For correct operation of quality configurable circuits, the following timing constraints have to be satisfied. First, to ensure completion of single cycle operations, Equation 2.13a constrains the delay of paths from inputs to outputs through SS_i. The terms $T_{cq}(I_{FF})$ and $T_{sp}(O_{FF})$ refer to C-Q delay and setup times of input and output FFs, respectively. Equation 2.13b and 2.13c ensure that the difference in substitutions are detected and the signals for EFF update and clock extension settle

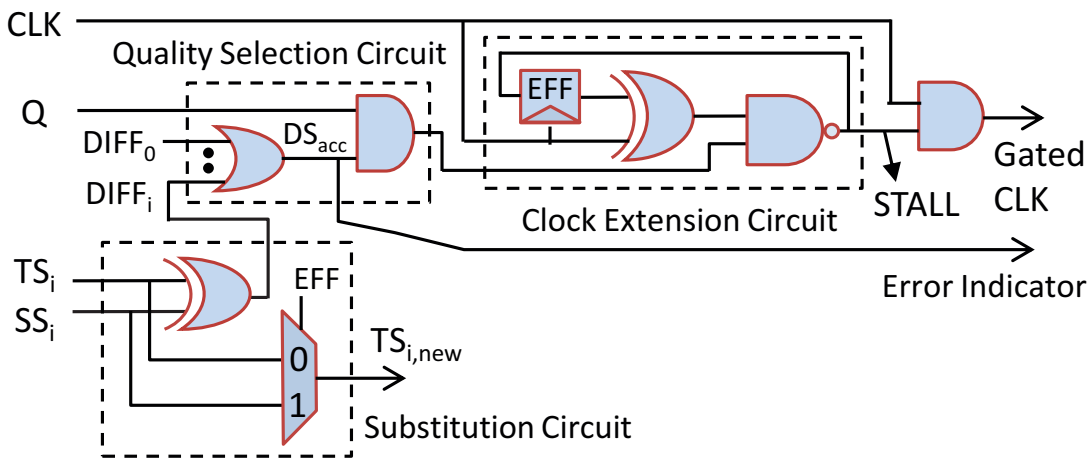


Fig. 2.29.: Selective substitution, quality selection and clock extension circuits

within the clock period. Finally, Equation 2.13d requires all paths originating from EFF to evaluate within the clock period, thereby ensuring two cycle operation.

$$T_{cq}\left(I_{FF}\right) + T_{max}\left(I \rightarrow SS_i \rightarrow O\right) + T_{sp}\left(O_{FF}\right) \leq T_{clk} \tag{2.13a}$$

$$T_{cq}\left(I_{FF}\right) + T_{max}\left(I \rightarrow TS_i \rightarrow STALL\right) + T_{sp}\left(E_{FF}\right) \leq T_{clk} \tag{2.13b}$$

$$T_{clk\_tr} + T_{cq}\left(I_{FF}\right) + T_{max}\left(I \rightarrow TS_i \rightarrow STALL\right) \leq T_{clk} \tag{2.13c}$$

$$T_{cq}\left(E_{FF}\right) + T_{max}\left(E_{FF} \rightarrow O\right) + T_{sp}\left(O_{FF}\right) \leq T_{clk} \tag{2.13d}$$

Note that clock skew is ignored in the above constraints for ease of explanation, but can be considered and addressed using conventional techniques.

The trade-offs involved in quality configurable synthesis are similar to approximate synthesis with one notable difference - since no logic is deleted, the potential for logic downsizing is critical. As shown in Figure 2.28, both the transitive fan-out and the independent logic of TS can be downsized as the timing constraint is relaxed at that point. However, additional area and power is consumed by the substitution, quality selection and clock extension circuits. Also, the occasional need for additional clock cycles in the accurate mode impacts performance. Our experiments demonstrate that, despite these overheads, by choosing proper substitution candidates, the energy

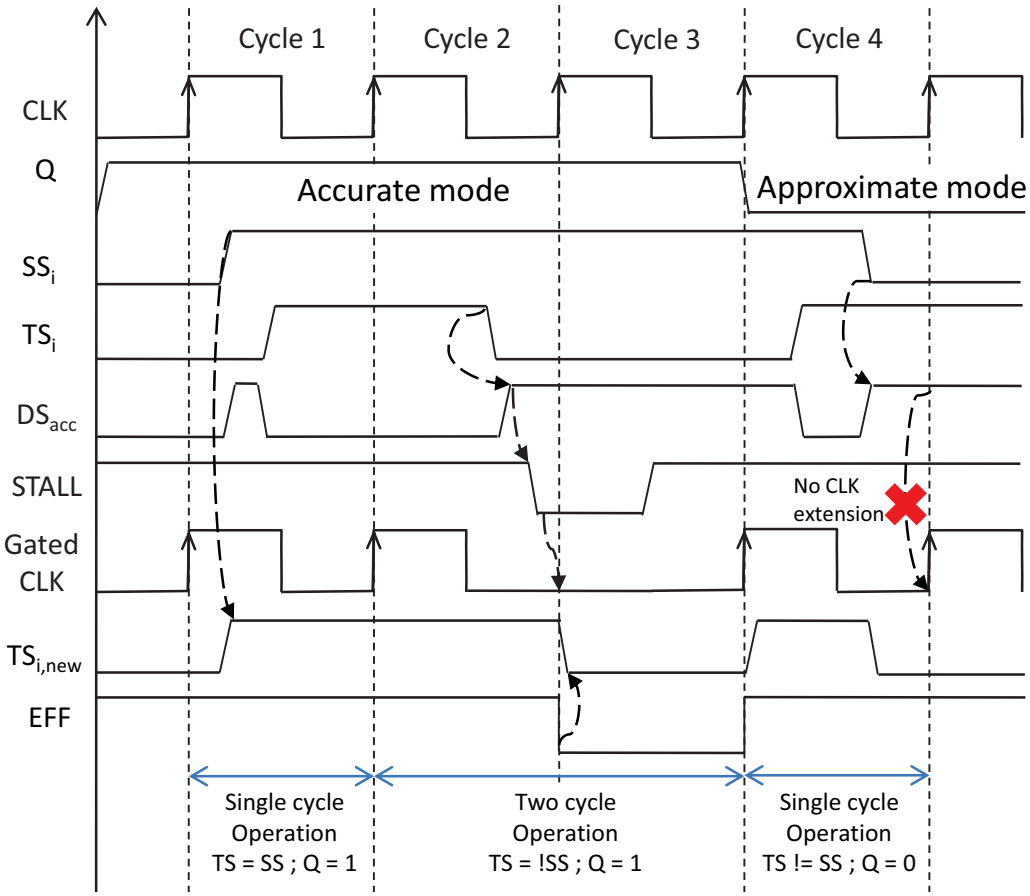


Fig. 2.30.: Timing diagram showing signal transitions in accurate and approximate modes

consumed is lowered even in the accurate mode. The energy savings are larger in the approximate mode, with no impact on performance. If the switch between quality modes is coarse grained, then the clock extension circuit can be power gated in the approximate mode, leading to additional savings in power/energy. Note that the actual error in the approximate mode is evaluated at the circuit outputs as before. The error indicator output, shown in Figure 2.28, is conservative since the difference in TS and SS may not be sensitized to the circuit's output.

In cases where multiple approximate modes exist, the substitutions are grouped such that an additional cycle is provided only to selected substitutions that cause intolerable errors. Thus from exclusive single cycle operation, the quality configurable

circuit progressively recovers from more and more errors (or substitutions) as the quality constraints are tightened.

2.4.3 SASIMI Methodology

This section details the automation of the proposed approach for approximate and quality configurable circuit synthesis.

Algorithm 2 Pseudo-code for SASIMI-Approximate

Input: Original Circuit: Ckt_{orig}, Target Error: ERR

Output: Approximate Circuit: Ckt_{approx}

1: **Begin**
2: Initialize: $NewCkt_{approx} = Ckt_{orig}$
3: Approximate Circuit Error: $ACE = 0$
4: **while** $ACE \leq ERR$ **do**
5: $Ckt_{approx} = NewCkt_{approx}$
6: $<TS,SS> = get\_substitution\_candidate(Ckt_{approx})$
7: $NewCkt_{approx} = substitute\ TS{=}SS$ in Ckt_{approx}
8: $Simplify\ (NewCkt_{approx})$
9: $ACE = compute\_error\ (Ckt_{orig}, NewCkt_{approx})$
10: **end while**
11: **return** Ckt_{approx}
12: **End**

Algorithm 2 describes the procedure for approximate circuit synthesis. The inputs to the algorithm are the original circuit (Ckt_{orig}) and the target error acceptable in its approximate implementation (ERR). The algorithm is iterative (lines 4-10) and performs successive substitutions until the imposed target error constraints are violated. In each iteration, the following steps are carried out: (i) the best candidate signal pair for substitution is identified (line 6), (ii) the substitution is performed (line

7) and the circuit is simplified (line 8), and (iii) the error in the approximate circuit is estimated (line 9). The algorithm proceeds to the next iteration if the approximate circuit error is less than the specified target error constraint (line 4). If not, the last legal approximate circuit is produced as the output (line 11).

The procedure used in the identification of candidate signal pairs for substitution is detailed in Algorithm 3. For each signal (S) in the given circuit, the following metrics are computed in lines 6-10. $S.TFI_{ind}$ (line 6) calculates the size of the independent logic that can be removed if S is substituted. This gives the measure of the logic deletion potential of S. $S.TFO_{crit}$ (line 8) is the size of the transitive fan-out of S, whose arrival times are constrained by it. This can be evaluated by setting the arrival time at S to be zero and recomputing the arrival times of its transitive fan-outs and identifying the gates whose arrival times have been relaxed. This, combined with the actual arrival time of S (maximum slack that can be introduced) gives the maximum potential for logic downsizing of S. A weighted sum of the normalized deletion and downsizing potentials is used to compute the signal score ($S.Score$) in line 10. The $S.Sens$ field (line 9) indicates if the signal fans-out to an output that cannot tolerate errors, in which case the $S.Score$ is made zero. Note that all the above fields can be computed efficiently in a single topological and reverse topological traversal of the circuit. The signals within a top fraction of the maximum score are designated as target signal TS candidates ($TS_{candidates}$).

Using the set $TS_{candidates}$, the best substitution pair is identified by lines 15-25 in Algorithm 3. For each signal in $TS_{candidates}$ and each legitimate substitute signal (SS) candidate, the substitution score ($SUB.Score$) is computed in line 19. $SUB.Score$ is similar to the signal score, except that the estimates of logic deleted and downsizing possible are refined. Further, the potential error introduced by the substitution (P_{diff}) is also taken into account. Using $SUB.Score$ as the metric, the best substitution pair is identified. It is worth noting that the computed $SUB.Score$ can be reused in the next iteration, if both TS and SS are unaffected during the simplify step, greatly reducing the runtime of Algorithm 3. The parameter α used to

Algorithm 3 Pseudo-code to obtain substitution candidate

Input: Circuit: Ckt

Output: Target-Substitute signal pair: TS_{best}, SS_{best}

Begin

Read Ckt and sort signals in topological order

A = Area of the circuit

D = Delay of circuit

for each S: Signals $\in Ckt$ **do**

$S.TFI_{ind}$ = Independent logic size in Tr.fan-in of S

$S.AT$ = Arrival time of S

$S.TFO_{crit}$ = Size of Tr.fan-out made critical by S

$S.Sens = 0$ if S fans out to a sensitive output; else 1

$S.Score = S.Sens * \left[\alpha(\frac{S.TFI_{ind}}{A}) + (1-\alpha)(\frac{S.AT}{D})(\frac{S.TFO_{crit}}{A})\right]$

end for

$TS\_max\_score = \max(S.Score) \quad \forall S \in Ckt$

$TS_{candidates} = \{S \ni (S.Score \geq \beta * TS\_max\_score)\}$

$SUB\_max\_score = 0$

for each $TS \in TS_{set}$ **do**

for each $SS \in \{0, 1, S \in Ckt \ni S.AT < TS.AT\}$ **do**

P_{diff}: Probability of $TS \neq SS$

$SS = (P_{diff} < 0.5)\ ?\ K : \overline{K}$

$SUB.Score = \left[\alpha\left(\frac{TS.TFI_{ind} - (TS.TFI_{ind} \cap SS.TFI_{ind})}{A}\right) + (1-\alpha)\left(\frac{TS.AT - SS.AT}{D}\right)\left(\frac{S.TFO_{crit}}{A}\right)\right] \Big/ P_{diff} * (1 - P_{diff})$

if $(SUB.Score > SUB\_max\_score)$ **then**

$SUB\_max\_score = SUB.Score$

$<TS_{best}, SS_{best}>$ = <TS, SS>

end if

end for

end for

return TS_{best}, SS_{best}

End

weight the logic deletion and downsizing potentials is chosen empirically based on the objective of the synthesis. In the case of approximate circuits, the deletion potential is given higher weight (larger α), since it is more significant that downsizing. In quality configurable synthesis, since the independent logic is retained, the downsizing potential carries more importance. For our experiments, α values of 0.75 and 0.25 were used for approximate and quality configurable synthesis, respectively.

Algorithm 4 Pseudo-code for SASIMI-Quality-Configurable

Input: Original Circuit: Ckt_{orig}, Error List: ERR_{list}

Output: Quality Configurable Circuit: Ckt_{qc}

1: **Begin**
2: Initialize: $NewCkt_{qc} = Ckt_{orig} + Clk\_Extension\_Ckt$
3: **for** each $ERR \in ERR_{list}$ **do**
4: **while** QC Circuit Error: $QCE < ERR$ **do**
5: $Ckt_{qc} = NewCkt_{qc}$
6: $<TS,SS> = get\_substitution\_candidate(Ckt_{qc})$
7: # Insert substitution circuit
8: # Add substitution to Quality selection circuit
9: $NewCkt_{qc} = form\_qc\_ckt\ TS{=}SS$ in Ckt_{qc}
10: $Simplify\ (NewCkt_{qc})$
11: $QCE = compute\_error\ (Ckt_{orig}, NewCkt_{qc})$
12: **end while**
13: **end for**
14: **return** Ckt_{qc}
15: **End**

Note that in choosing the best substitution candidate for a given circuit, the potential benefits in terms of logic deletion and downsizing are weighed against the error introduced by the substitution. Hence, although the substitutions are chosen in a greedy manner within each iteration of Algorithm 2, the use of benefits-to-error

ratio (instead of choosing just based on the benefits) in the selection process implies that the algorithm does not entirely exhaust the error constraints in a given iteration but rather retains some flexibility for future iterations.

Algorithm 4 describes the methodology for quality configurable synthesis. A list of error constraints corresponding to the desired quality modes is provided as input. For each quality mode (lines 3-12), the procedure resembles approximate synthesis (Algorithm 2), except that in addition, the substitutions are grouped based on the quality mode in the quality selection circuit (line 9). The substitution and clock extension circuits are also appropriately added to the circuit. Note that, in the proposed procedure, the set of substitutions causing two cycle operation in a given quality mode is a strict super set of all modes with lower quality constraints. This greatly simplifies the design of the quality selection circuit.

Using Algorithms 2, 3 and 4, the SASIMI paradigm can automatically synthesize approximate and quality configurable circuits for any given circuit and quality constraint.

2.4.4 Experimental Methodology

We evaluated SASIMI on a wide range of benchmarks including (i) arithmetic circuits *viz.* adders - Kogge stone (KSA), Ripple carry (RCA), Carry lookahead (CLA), multipliers - Wallace tree (WTM), array (MUL) and Multiply and Accumulate (MAC), (ii) complex datapath modules *viz.* Sum of Absolute difference (SAD), Euclidean Distance unit(EUDIST), FFT Butterfly (BUT), and (iii) circuits from the ISCAS85 benchmark suite. Figure 2.31 shows the CAD flow adopted in the implementation of SASIMI. The benchmarks were synthesized using Synopsys Design Compiler Ultra [34] and mapped to the IBM 45nm technology library. The metrics used for identifying substitution candidates were computed using custom tools that analyzed the synthesized netlist. The signal probability calculation engine in Synopsys Power Compiler was used to obtain difference probabilities among signal

pairs. For the experiments, all input vectors were considered equi-probable and the synthesized approximate and quality configurable circuits were evaluated for area and power at iso-delay. In the case of quality configurable circuits, the additional hardware overheads in substitution, clock extension and quality selection circuits are included during area and power estimation, and an energy comparison is performed taking into account the additional clock cycles incurred.

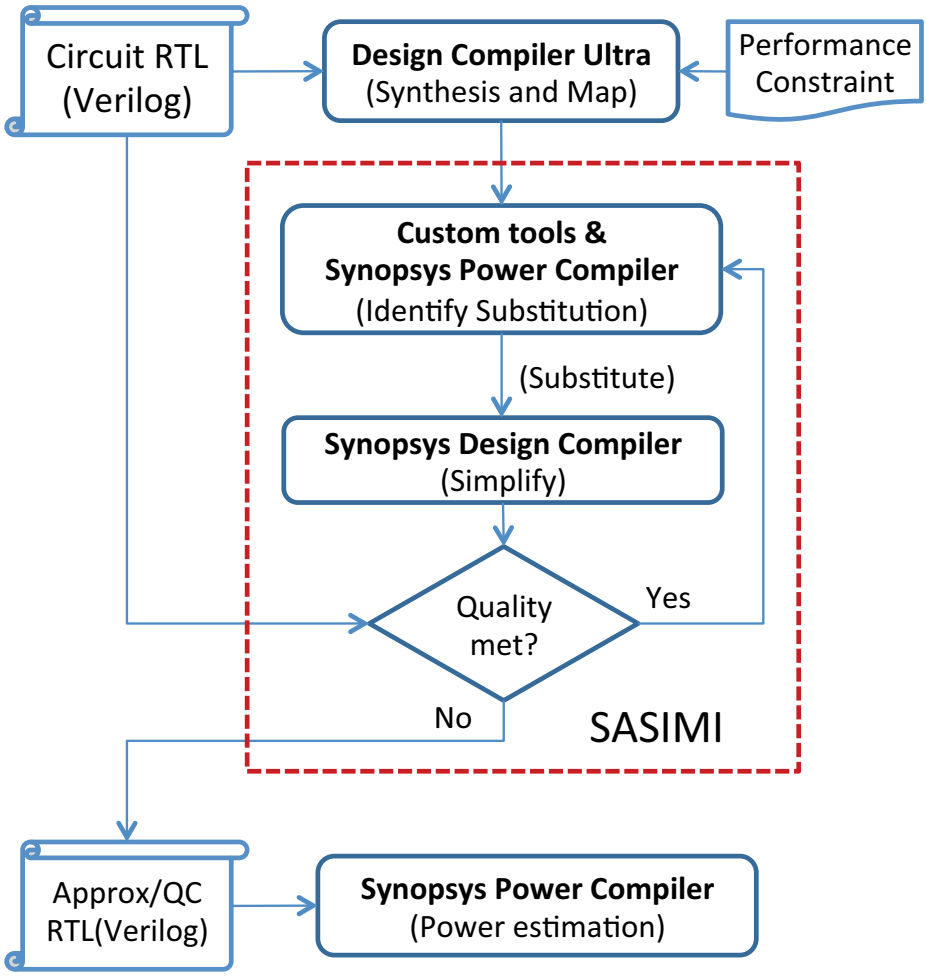


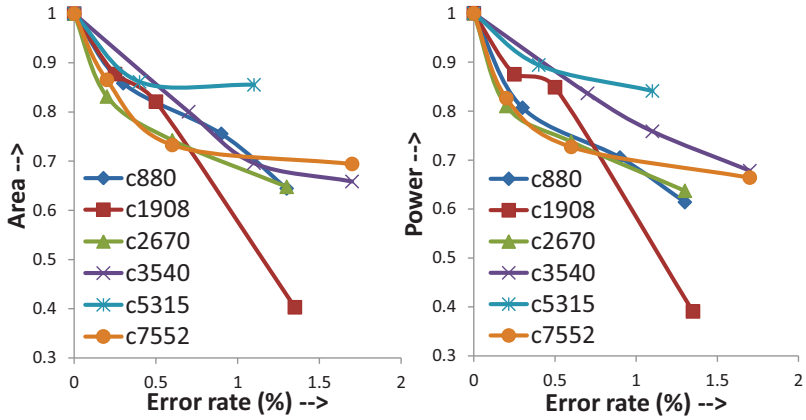
Fig. 2.31.: CAD flow employed in the implementation of SASIMI

The circuits were synthesized for two different target quality metrics *viz.* error probability and average error magnitude described in Section 2.2. The error probability metric was estimated by ORing the difference (O_{orig} xor O_{approx}) at all output

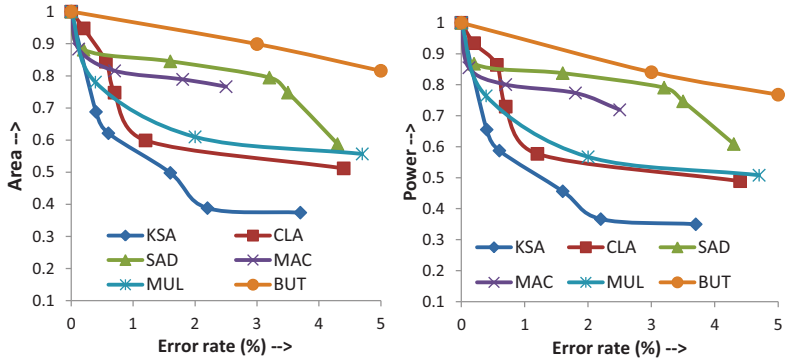
bits and then computing its static signal probability. On the other hand, the average error magnitude metric was evaluated by summing the difference signal probabilities of all output bits weighted by their numerical significance.

2.4.5 Results: Approximate Circuits

This section presents the results of various experiments that compare circuits generated by SASIMI to well-optimized, accurate baselines.



(a) ISCAS85 benchmarks



(b) Arithmetic modules

Fig. 2.32.: Area and power benefits for error probability metric

We begin by comparing the area and power consumed by the approximate circuits relative to the original circuit at iso-delay, for a range of error probability and average

error magnitude constraints. In the case of error probability metric, as shown in Figures 2.32a and 2.32b, we see that as the error constraints are relaxed, significant benefits are obtained across all circuits. For ISCAS85 benchmarks, we see significant benefits that amount to 15%-25% in area and 10%-28% in power are achieved for tight error probabilities (less than 0.5%). As the error constraints are relaxed (less than 2%), area and power improvements between 30%-60% are obtained. For arithmetic circuits, the improvements in area and power range from 15%-40% for tight error probabilities (less than 1%) and upto 65% for relaxed error probabilities (less than 5%). In the case of ISCAS benchmarks, the benefits amount to 15%-35% in area and 10%-40% in power for tight error probabilities and upto 60% for relaxed error probabilities.

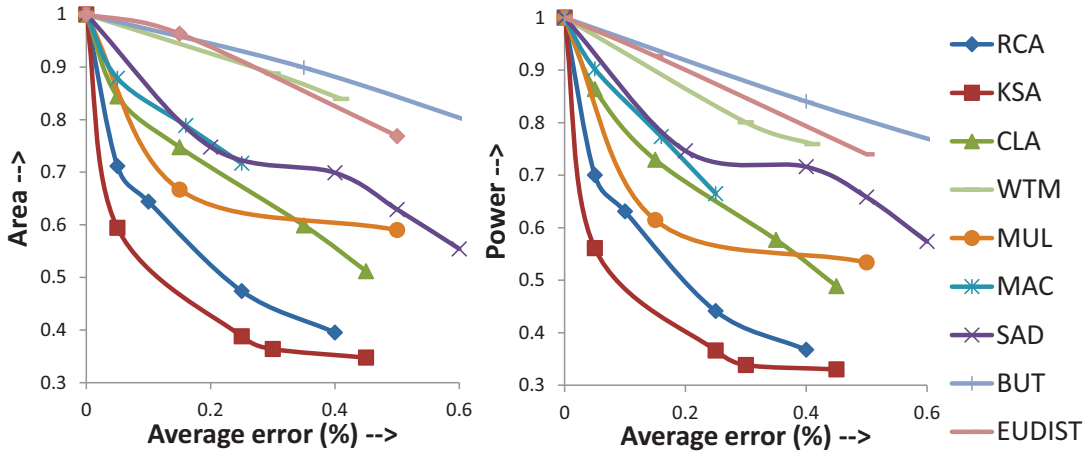


Fig. 2.33.: Area and power benefits for average error magnitude metric

Similar trends are observed for the average error magnitude metric. As depicted in Figure 2.33, for average errors of less than 0.5% of the maximum value, improvements in the range of 15%-65% in area and 20%-68% in power are obtained. These results demonstrate the applicability and effectiveness of the SASIMI design approach and synthesis methodology.

To demonstrate the effectiveness of functional approximations across the entire design space, the approximate circuits were synthesized for a range of delay val-

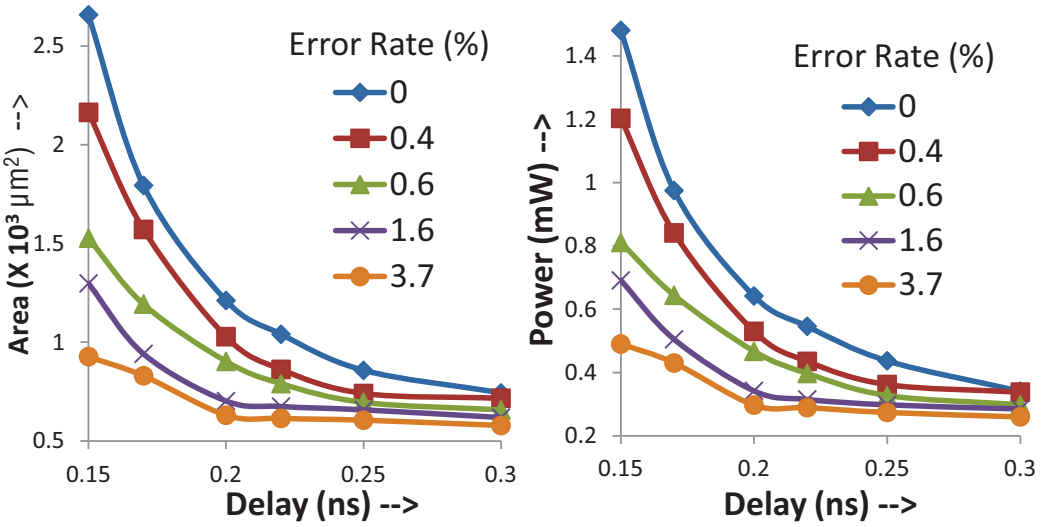


Fig. 2.34.: Area and power of KSA with delay sweep

Table 2.3.: Quality configurable circuits synthesized for average error magnitude with two quality modes

Circuit	Delay (ns)	Area (%)	Power (%)	P_{2cycle}	$Energy_{acc}$ (%)	$Energy_{app}$ (%)	Avg.Error (%)
RCA	0.65	16.2	21.6	0.24	2.69	36.7	0.03
CLA	0.17	29.5	26.72	0.067	21.8	34.9	0.01
MAC	0.55	13.7	16.8	0.014	15.6	18.3	0.01
EUDIST	0.47	24.1	22.24	0.0075	21.6	24	0.12
MUL	0.35	32.9	36.5	0.05	33.3	40.05	1.2
SAD	0.5	11.6	12.1	0.002	12.0	14.44	0.01

ues. Figure 2.34 shows the area and power *vs.* delay plots for a 32-bit Kogge stone adder with error probability metric. The approximate circuit outperforms the original circuit at all delay values, which testifies to the wide scope of the optimizations performed by SASIMI.

Table 2.4.: Quality configurable circuits synthesized for error probability metric with two quality modes

Circuit	Delay (ns)	Area (%)	Power (%)	P_{2cycle}	$Energy_{acc}$ (%)	$Energy_{app}$ (%)	$ErrProb$ (%)
KSA	0.2	16.3	14.79	0.009	14.02	22.27	0.7
c880	0.22	13.1	18.03	0.064	12.78	31.9	4.8
c1908	0.25	13.8	22.9	0.0102	22.11	31.31	0.95
c2670	0.22	5.09	15.68	0.0051	15.25	24.51	0.2
c3540	0.36	21.94	19.72	0.008	19.08	23.56	0.65
c7552	0.32	12.79	19.18	0.064	14.01	22.54	4.8

2.4.6 Results: Quality configurable circuits

Next, we present the results obtained for quality configurable designs with two quality modes synthesized by SASIMI.

Table 2.3 tabulates the area, power and energy improvements obtained relative to the original circuit for the average error magnitude metric. Column 3 shows the percentage reduction in area. Despite the area overheads of the additional circuits, the overall area savings range from 13%-32% compared to the original circuit. These benefits come from the logic downsizing that was possible by the substitutions. Column 4 provides the corresponding improvement in power. The probability of two cycle operations in the accurate mode is listed in Column 5. The total energy savings in the accurate and approximate modes are listed in Columns 6 and 7. In the accurate mode, the circuit recovers from all errors, and the energy benefits are due to the savings from logic downsizing outweighing the overheads of the added control circuitry and two cycle operation. In the approximate mode, the additional energy savings stem from two sources - the circuit has fewer or no two cycle operations as it does not recover from all errors, and the clock extension circuit could be power gated in the lowest quality mode. Due to these reasons, net energy savings between

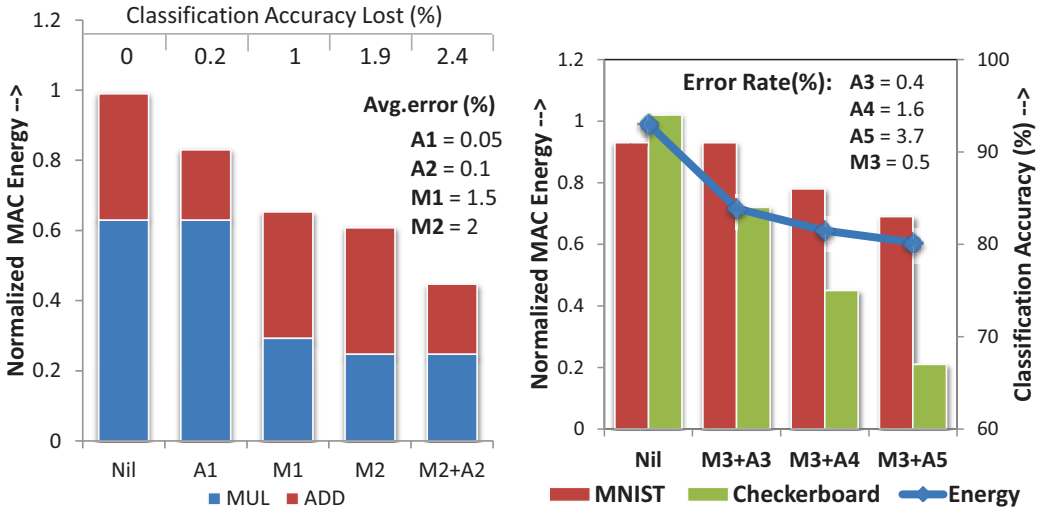
14%-40% are obtained. Finally, column 8 gives the average error magnitude in the approximate mode of the circuit.

Table 2.4 reports similar results for circuits synthesized using the error probability metric. Energy benefits in the range of 22%-32% are obtained in the approximate mode for error probabilities less than 1%. Note that the rate of two cycle operations is a little larger than the actual error probability at the outputs, because not all errors at the substitution points are sensitized at the outputs. In other words, the clock extension mechanism and the error indicator output bit are conservative. In summary, our experiments suggest that SASIMI is a promising approach for the synthesis of approximate and quality configurable circuits.

2.4.7 Application-level evaluation of SASIMI circuits

We utilized the circuits synthesized by SASIMI to build a 25 X 25 systolic array architecture [7] and analyzed the impact of approximations at the application level. We chose applications based on two widely used classification algorithms *viz.* Support Vector Machines (SVM) and K-Nearest Neighbors (k-NN). Approximate adders (A1 to A5) and multipliers (M1, M2, M3) synthesized with SASIMI were used in the multiply and accumulate units of the systolic array. Figure 2.35 illustrates the resulting energy *vs.* classification accuracy trade-off obtained at the application level.

In case of k-NN, as shown in Figure 2.35a, for nearly no loss in accuracy ($\leq 1\%$), we obtain upto 30% energy savings in the MAC units. Experiments on the SVM algorithm were conducted using two datasets viz. MNIST Digit recognition [39] and Checkerboard. Figure 2.35b shows the energy gained for several accuracy levels in each case. We observe that, for MAC units of similar error and energy profiles, the loss in classification accuracy for the Checkerboard dataset is substantially larger than MNIST. This proves the sensitivity of application resilience to the dataset being processed, underscoring the need for quality configurability at runtime. Further,



(a) kNN-Checkerboard dataset

(b) SVM

Fig. 2.35.: Energy *v.s.* accuracy trade-off in classification

these experiments demonstrate that significant energy benefits could be harnessed by utilizing the synthesized computation units in domain specific architectures.

2.5 Summary

A promising approach to approximate computing in hardware is through the design of approximate and quality configurable circuits that are highly efficient in their implementation, while meeting the user-specified quality requirements in evaluating the required function. However, one of the key challenges impeding their mainstream adoption is the lack of systematic design methodologies that are general and scalable to any given circuit and quality constraints. Towards this objective, this chapter described two approaches *viz.* SALSA and SASIMI that allowed automatic design and synthesis of approximate and quality configurable circuits. The first approach SALSA transforms the problem of approximate circuit synthesis into a well studied logic synthesis problem and identifies don't care conditions in the circuit that are borne out of the quality specifications in order to simplify its implementation. On the other hand,

SASIMI adopts a circuit transformation approach, wherein the key idea is to identify substitutions in the circuit that foster logic deletion and downsizing (simplification) while introducing minimal error. Given a user specified quality requirement, the substitution and simplification steps are iteratively performed while the error constraints are satisfied. SASIMI was further extended to synthesize quality configurable circuits, where at runtime, selected input vectors were given an additional cycle to correct errors due to approximations. The approaches were utilized to synthesize approximate and quality configurable versions of a wide range of benchmarks including arithmetic circuits, complex blocks and entire datapaths using different quality metrics, demonstrating generality and significant benefits in energy and area. In addition, the utility of the techniques were reaffirmed through case studies evaluating the benefits of the synthesized approximate circuits at the application-level.

In summary, the design techniques and synthesis methodologies presented in this chapter promise a new frontier in the design automation of approximate and quality configurable circuits.

3. QUALITY PROGRAMMABLE PROCESSORS

3.1 Introduction

The previous chapter presented systematic methodologies for designing the basic building blocks of approximate computing *viz.* approximate and quality configurable circuits. Several research efforts have utilized such techniques to design larger approximate computing systems at higher levels of design abstraction [5, 7–13, 40]. These techniques have adequately demonstrated the benefits of approximate computing, but almost always in an application-specific context. However, the mainstream adoption of approximate computing, and its use in a broader range of applications, are predicated upon the creation of programmable platforms for approximate computing.

A key requirement for programmable approximate computing platforms is that they should provide a suitable HW/SW interface using which (i) programs can naturally expose inherent application resilience, and (ii) hardware designers can independently develop techniques to leverage the flexibility that it engenders. However, this is not easily achieved. Previous attempts [11, 13, 14, 40] to extend approximate computing to the realm of programmable processors have resulted in only small energy benefits as they are limited on two key fronts. First, their HW/SW interface identifies computations that can be subject to approximations without constraining the quantity of error that can be tolerated during execution. As shown in Section 3.2, for a given application output quality requirement, allowing arbitrary errors during execution severely limits the fraction of instructions to which approximations can be applied. Second, they target applying approximate computing to complex general-purpose cores. This fundamentally restricts their energy benefits, since the energy consumption of such cores is dominated by control front-ends such as instruction fetch, decode, dispatch, retire *etc.* that are inherently not amenable to approximation.

This chapter extends the state-of-the-art in approximate computing in two important directions.

- It proposes the concept of *quality-programmable processors*, wherein software expresses its tolerance for approximate computing at the natural hardware-software interface, *i.e.*, the instruction set. The two key ingredients of a quality programmable processor are (i) a *quality programmable ISA* (QP-ISA), in which instructions are associated with quality fields that explicitly indicate the desired accuracy level that must be *guaranteed* during their execution, and (ii) a *quality programmable micro-architecture* (QP-uArch), which is equipped with hardware mechanisms to translate instruction-level quality specifications into improvements in energy or performance. As a further enhancement, quality programmable processors may provide feedback to software regarding the actual error incurred during the execution of an instruction. Software may use this information to modulate the quality specifications for future instructions.

- It explores approximate computing in the context of a new class of programmable architectures, *viz.* vector processors. Driven by the characteristics of a variety of resilient workloads and considerations unique to approximate computing, we propose Quora, a 1D/2D vector processor built from the ground up for approximate computing. Quora is designed with a 3-tiered hierarchy of processing elements – Approximate Processing Elements (APE), Mixed Accuracy Processing Elements (MAPE), and Completely Accurate Processing Element (CAPE) – that provide distinctly different energy *vs.* quality trade-offs. We propose hardware mechanisms based on precision scaling with error monitoring and compensation to facilitate quality-configurable execution on these processing elements and demonstrate significant energy benefits.

The rest of the chapter is organized as follows. Section 3.2 motivates the need for quality programmability in the instruction set. Section 3.3 provides an overview of quality programmable processors and their key concepts. Section 3.4 describes

the instruction set and micro-architecture of the proposed quality programmable vector processor, QUORA. Section 3.5 outlines the hardware mechanisms employed in QUORA to enable quality-configurable execution. Section 3.6 explains the evaluation methodology, and the results of experiments conducted on a suite of benchmark applications are presented in Section 3.7. Finally, Section 3.8 concludes the chapter.

3.2 A Case for Quality Programmability

A key aspect of quality programmable processors is that their HW/SW interface allows software to not just identify approximate instructions, but to also explicitly specify the quantity of error that each instruction can tolerate during its execution. Previous efforts to extend approximate computing to programmable processors [11,13,14,40] only denote instructions as accurate or approximate, while allowing errors of arbitrary magnitude to occur in approximate instructions. We motivate the need for quality programmability in instructions through two representative applications — image segmentation and handwritten digit recognition. The applications were compiled to an x86 platform and an error injection framework, similar to [4], was utilized to identify resilient instructions and inject errors in their outputs. The errors were random but constrained to be within a pre-specified instruction-level quality bound. With this setup, for different application-level quality requirements, the instruction-level quality bounds in the program are varied to study its impact on the number of dynamic instructions that can be approximated.

Figure 3.1 shows the fraction of each application's dynamic instructions that may be executed approximately (Y-axis) *vs.* the loss in application output quality (X-axis) for different levels of instruction accuracy. We observe in both cases that, when the magnitude of error in the execution of approximate instructions is bounded, there is a large increase in the percentage of dynamic instructions that may be approximated for any given application output quality. For example, consider point A in the graph for the image segmentation application, where the application-level quality loss is 2%.

In this case, when the instruction-level accuracy is constrained to within 2.5% of the maximum value, the fraction of instructions that can be approximated increases by 16X compared to when errors of arbitrary magnitude are allowed. For this application, across different application quality requirements, we observe between 10X-30X increase in the number of approximate instructions when instruction-level errors are bounded to within 2.5% of the maximum value. Similarly, in the case of handwritten digit recognition, the fraction of dynamic instructions that may be approximated increases by 25X-107X when the instruction-level errors are constrained to less than 7.5% of the maximum value. Since the fraction of application instructions that may be approximated is a key determinant on the energy savings that approximate computing can provide, we conclude that the capability to provide quality-programmability in hardware is highly desirable.

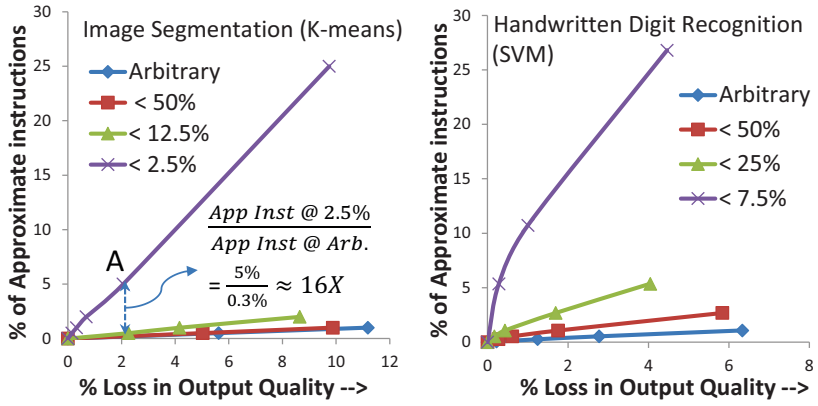


Fig. 3.1.: Fraction of instructions that may approximated under arbitrary *vs.* controlled approximations

3.3 Quality Programmable Processors: Concept & Overview

To enable broader use of approximate computing techniques in a programmable context, we introduce the concept of *quality programmable processors* (QPPs), illustrated in Figure 3.2, in which: (i) software has the ability to express application

resilience as accuracy bounds/expectations at the outputs of individual instructions, and (ii) hardware is equipped to understand and guarantee these accuracy bounds, while exploiting the flexibility that they provide to obtain energy savings. This section provides a conceptual overview of QPPs and their key components.

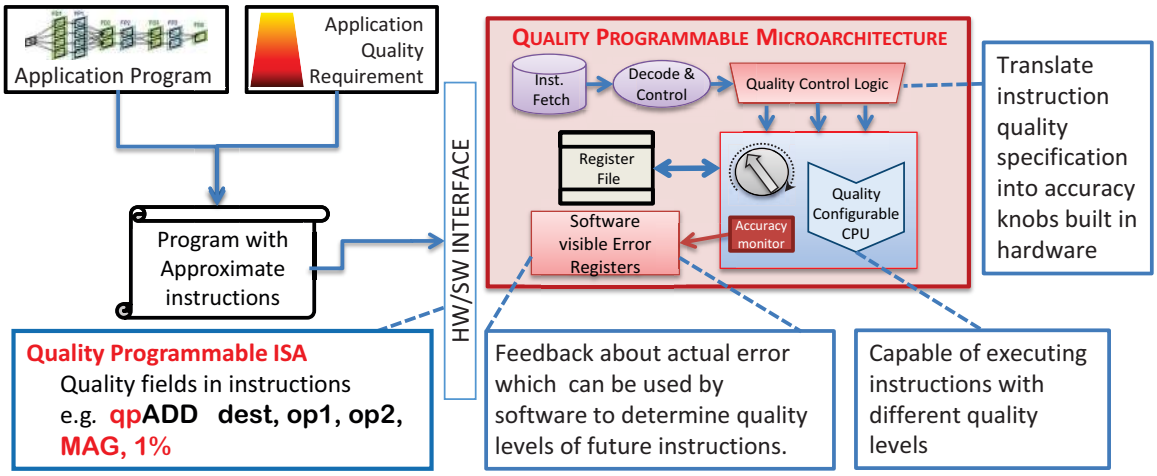


Fig. 3.2.: Conceptual overview of a quality programmable processor

3.3.1 QP-ISA: Quality Programmable ISA

The primary requirement of QPPs is to provide software with the ability to expose application resilience to hardware. We propose to express resilience in terms of accuracy bounds or expectations at the outputs of individual instructions through a quality programmable instruction set architecture (QP-ISA). Selected instructions in the ISA are extended with *quality fields* to indicate the desired level of accuracy with which the instruction output must be computed. The quality fields denote both the type and amount of error that can be tolerated during execution of the instruction. The error may be expressed as a strict bound or as an expectation over all possible inputs. For example, the *add* instruction shown below requires the output error magnitude to be within 1% of the maximum output value.

$$\textbf{qpADD} \quad dest,\ op1,\ op2,\ ERR.MAG,\ 1 \tag{3.1}$$

Note that the quality specifications are based purely on instruction semantics, and do not depend on the specific approximate computing design technique employed in hardware. This allows software to remain oblivious of hardware design techniques and fosters portability across different quality programmable hardware platforms.

In summary, a quality programmable ISA contains quality programmable instructions, which are enhanced with explicit quality fields to indicate the desired accuracy level required during execution. As an example of QP-ISA, the instruction set of the proposed quality programmable vector processor, QUORA, is discussed in Section 3.4.

3.3.2 QP-uArch: Micro-architecture with Accuracy-Energy Trade-off

The micro-architecture of a quality programmable processor contains execution units whose computational accuracy is dynamically configurable at runtime. In other words, these execution units possess accuracy knobs in hardware that can be modulated to trade-off accuracy for energy consumption. A variety of approximate computing design techniques may be employed for this purpose. For example, voltage over-scaling is a popular technique where the supply voltage is reduced to benefit energy while sacrificing accuracy due to errors caused by timing violations.

The micro-architecture also contains a *Quality control unit* that is capable of understanding the quality fields present in instructions and translates them into hardware accuracy knobs, such that quality requirement dictated by the instruction is guaranteed to be met. The quality translation may be static, where the hardware accuracy knobs are set once at the start of instruction execution, or dynamic, in which case, the instructions typically execute over multiple cycles and the knobs are varied during the course of execution.

3.3.3 Quality Monitors: Error Feedback to Software

While a QPP guarantees that the accuracy levels set in the instruction, the error that occurs during the actual execution may be relatively low compared to the specification. This can be attributed to the following reasons: (i) In case of most approximate design techniques, the error introduced in computation is input dependent. When errors adhere to pre-determined worst-case bounds, their variance across all inputs is significant. (ii) The quality translation process is typically conservative in order to guarantee the quality bounds without incurring excessive overheads in the quality control unit.

We instrument the micro-architecture with *quality monitors* that estimate the amount of error at the output of an instruction. This error estimate is provided back to the software through special *error registers*. The ISA is enhanced with instructions to access and operate on these error registers, which can be then be used to decide the quality levels of future instructions. This error feedback enables software programs to be robust to varying input data characteristics and differing approximate design techniques employed in hardware.

3.3.4 Programming QPPs

In QPPs, the responsibility of maintaining the application-level output quality is shared between hardware and software. The application developers, in addition to the application source code, provide a quality requirement expected at the output of the application. The quality requirement is application-specific and is typically based on the context in which the application is being used. The application-level quality is then translated into accuracy requirements at the outputs of instructions. Previously proposed analysis and profiling frameworks such as [4, 41–43] can be employed for this purpose. The hardware exploits the flexibility provided by the relaxed instruction accuracy to maximize energy savings subject to meeting the quality constraints dictated by software. In addition, the interface allows hardware to dynamically provide feed-

back on the actual error incurred during execution, which can then be utilized by software to relax the quality specifications of future instructions more aggressively.

3.4 QUORA: A Quality Programmable Vector Processor

As a first embodiment of quality programmable processors, we describe a quality programmable vector processor, QUORA, which is designed to execute a intrinsically resilient applications from various domains including recognition, mining, synthesis, video processing, search *etc.* This section outlines the key design philosophy of QUORA, and describes in detail its instruction set and micro-architecture, which are driven by characteristics of the target applications as well as considerations unique to approximate computing.

In quality programmable designs, the energy savings stem predominantly from scaling the accuracy of execution units in the processor. The control front-ends, such as instruction fetch and decode, inherent to any programmable processor, have to be performed in an accurate manner. Thus, *the energy benefits from approximate computing are limited by the fraction of energy consumed by execution units in the design.*

Keeping in mind the above observation, and the fact that the application domains of interest invariably contain significant fine-grained data parallelism, we explore approximate computing in the context of a new class of programmable architectures, *viz.* vector processors. While vector processors intrinsically amortize the cost of control front-ends over several execution units, we further amplify this benefit in QUORA by employing a 3-tiered hierarchy of processing elements – a 2D array of Approximate Processing Elements (APE), 1D arrays of Mixed Accuracy Processing Elements (MAPE), and a scalar Completely Accurate Processing Element (CAPE) – that are characterized by distinctly different control complexity and energy *vs.* quality trade-offs. Moreover, since the instructions operate on data vectors (or streams), their execution typically extends over a large number of cycles, further reducing the

contribution of control front-ends to the overall application energy. Finally, the vector instructions in QUORA also provide an added advantage in that instruction-level quality specifications correspond to coarse-grained computations rather than individual scalar operations. The micro-architecture propagates errors across the scalar operations within a vector instruction and ensures that the instruction-level quality specifications are satisfied.

3.4.1 QUORA Instruction Set

This section describes the quality programmable instruction set of QUORA. In order to determine the right set of instruction primitives, we analyzed a wide range of resilient applications by hierarchically breaking down the computations involved as shown in Figure 3.3. We identified that these applications typically operate on data streams and perform matrix-matrix and matrix-vector operations, generating significant intermediate data. These intermediate results are then subject to complex reduction operations to produce a small number of outputs. Leveraging these characteristics, the proposed quality programmable instruction set is designed with 2D and 1D vector instructions, balancing efficiency and programmability.

Software Visible Micro-architectural State

Figure 3.4 shows the software visible micro-architectural state in QUORA, which consists of the following components:

- **Streaming Memory Bank:** QUORA contains 2 streaming memory banks, each containing streaming memory (SM) elements that are First-in-First-out (FIFO) buffers of equal length. Each SM element can be individually addressed and data vectors of length less than or equal to its size can be read or written. To facilitate data re-use, the internal *read* and *write* pointers of SMs can be altered relative to their current position through special instructions.

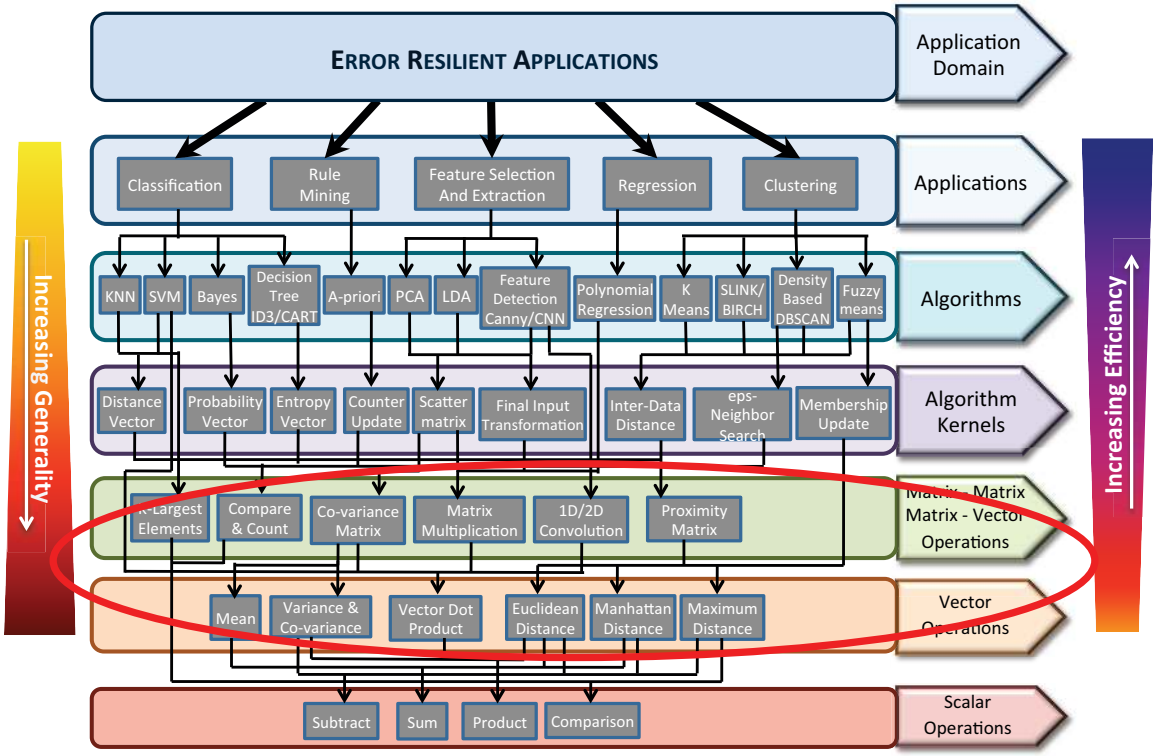


Fig. 3.3.: Resursive breakdown of computations in error resilient applications

- **2D-array accumulator registers:** QUORA features a 2-dimensional array of processing elements, each containing an accumulator register. Data can be scanned in/out of the accumulators in the 2D-array by row or column.

- **1D-array accumulators, Register files and Mask:** Two 1-dimensional arrays of processing elements are arranged along the top and left borders of the 2D array. The 1D-array elements in a given set operate together in a Singe-Instruction-Multiple-Data (SIMD) fashion. Each element contains an accumulator register, a small register file, and a mask register that can be set or reset through special instructions.

- **Scalar Register file:** In addition to the 2D and 1D processing elements, QUORA has a scalar processor with a register file random-accessible by software.

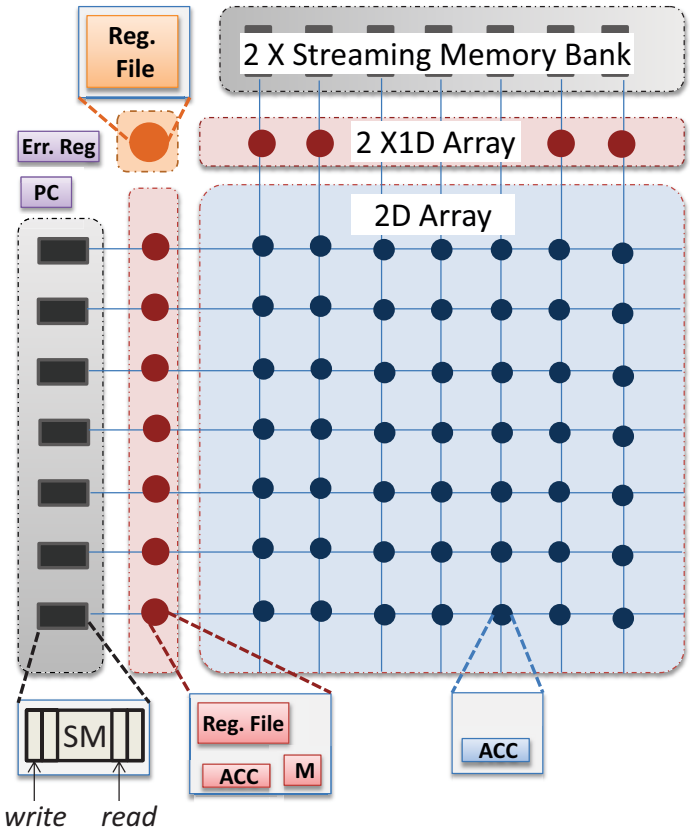


Fig. 3.4.: Software visible micro-architectural state in QUORA

- **Error Register:** The error register holds an estimate of the actual error incurred during the execution of quality programmable instructions. This register can only be read (but not modified) by software.
- **Program Counter/Status Register:** The status register contains flags that denote status of the processor.

Quality Programmable Instructions in QUORA

Quality programmable instructions allow software to explicitly indicate the error tolerable during their execution. In QUORA's ISA, instructions contain two quality fields that denote the type and amount of error. The type of error may be a strict bound or an expectation over all possible inputs. The error types supported in the

QUORA ISA are defined below. In these definitions, O_{acc} and O_{approx} denote the accurate and approximate outputs of the instruction.

- **Maximum Error Magnitude**, shown in Equation 3.2, bounds the absolute difference between the accurate and approximate outputs of an instruction to be less than a specified threshold.

$$MaxErr = MAX_{\forall inputs}(|O_{acc} - O_{approx}|) \tag{3.2}$$

- **Average Error Magnitude**, shown in Equation 3.3, an expectation of the absolute error magnitude over all possible inputs to the instruction.

$$AvgErr = \frac{\sum_{\forall inputs} |O_{acc} - O_{approx}|}{Total\ no.\ of\ Inputs} \tag{3.3}$$

- **Error probability**, shown in Equation 3.4, bounds the fraction of inputs for which the instruction can produce an incorrect value.

$$ErrProb = \frac{No.\ of\ Inputs\ for\ which\ O_{acc} \neq O_{approx}}{Total\ no.\ of\ Inputs} \tag{3.4}$$

Instruction Types

QUORA features a rich instruction set architecture containing 47 instructions - 13 scalar, 9 2D-array , 22 1D-array, 3 SM - that execute on the respective processing elements. The ISA is classified into 6 instruction types and Figure 3.1 tabulates a subset of most commonly used instructions in each category. The different instruction types are described below.

Scalar Instructions: The scalar instructions are similar to instructions in a RISC ISA. The operands are fetched from either the scalar register file or the error register and the result is always stored back to the scalar register file. They do not carry any quality fields and are always required to execute in an accurate manner.

Streaming Memory Instructions: These instructions are used to load data vectors into the streaming memory (SM) elements. The operands to the instructions

Table 3.1.: Representative instructions in QUORA's ISA

Inst. Type	Instruction	Function
Scalar Instructions	**LDRI** *Rd, value*	Load scalar register with immediate value
	ADDR *Rd,Rs1,Rs2*	Add scalar registers: Rd <= Rs1 + Rs2 *Error register* can be an source operand
	BEZ *Rs, Rel. address*	Branch if register Rs is zero
	HALT	End of program
Streaming Memory instructions	**LDSM** *R_length, stride, burst, R_st_add, R_row_enb, R_col_enb*	Load streaming memories indicated in *R_row_enb*, *R_col_enb* with data values from address *R_st_add*. *R_length* elements are fetched in bursts of size *burst*. Address is incremented by *stride* between bursts.
2D Array Instructions	**qpMAC** *R_length, R_row_enb, R_col_enb, R_q_type, R_q_amt*	Quality programmable dot product of rows and columns indicated by enable registers
	qpMOD2 *R_length, R_row_enb, R_col_enb, R_q_type, R_q_amt*	Quality programmable L2 Norm of rows and columns indicated by enable registers
	STR *<r/c>, R_stride, R_burst, R_st_add, R_row_enb, R_col_enb*	Store selected 2D array accumulator registers to memory in row/col. fashion
	RSTPE *R_row_enb, R_col_enb*	Reset accumulators
1D Array Reduction Instructions	**qpACC** *<r/c>, R_row_enb, R_col_enb, R_q_type, R_q_amt*	Quality programmable accumulation of selected values stored in the 2D-array accumulator registers in row/col. fashion
	qpMIN *<r/c>, R_row_enb, R_col_enb, R_q_type, R_q_amt*	Quality programmable Min. of selected values stored in the 2D-array accumulator registers in row/column fashion
1D Array Streaming Instructions	**SEQ** *R_length, SReg, R_row_enb, R_col_enb*	Checks if input operand from SM equals the value in its register *SReg*. If so, SM is set to 1
1D Array Self-Operand Instructions	**MVASR** *<r/c>, R_<r/c>_enb, SReg*	Move value in register *SReg* to accumulator
	qpADDX *<r/c>, R_<r/c>_enb, SReg, R_q_type, R_q_amt*	Quality programmable extended addition of registers *<SReg+1,SReg>* to accumulator
	qpMUL *<r/c>, R_<r/c>_enb, SReg, R_q_type, R_q_amt*	Quality programmable multiplication of accumulator with register *SReg*
	STMCG *<r/c>, R_<r/c>_enb, SReg*	Set Mask register is accumulator greater than register *SReg*
	INVM *<r/c>, R_<r/c>_enb*	Invert mask register

are fetched from the scalar register file. The start address is present in register *R_start_add*. The elements of the data vector are fetched in a burst-stride fashion, where *burst* number of data elements are loaded from contiguous address locations after which the data address is incremented by *stride*. This pattern is repeated until all elements of the data vector are loaded into the SM. Multiple streaming memory elements, indicated in the register *R_enb_stream*, can be loaded simultaneously with the same data. Instructions to manipulate the internal *read* and *write* pointers of the SMs also belong to this class.

2D-array Reduction Instructions: These instructions execute on the 2D-array of processing elements, and typically perform reduction operations (*e.g.,* Multiply-and-accumulate) on the data vectors present in the SMs. Thus, in a single instruction, all data vectors present in row SMs are operated with all data vectors in the column SMs. These instructions extend over variable numbers of cycles based on control parameters such as the vector length (*R_length*), rows (*R_row_enb*) and columns (*R_col_enb*) of the 2D-array that are active during execution. These instructions are quality programmable and contain quality fields, whose values are obtained from the scalar registers *R_q_type* and *R_q_amt*.

1D-array Reduction Instructions: Instructions of this type perform single-vector reduction operations (*e.g.,* min/max of a vector) and are executed on the PEs present in the 1D-array. The PEs obtain their data operands by either scanning out the accumulator registers of the 2D-array in row/column fashion or from the streaming memory element located in their corresponding row/column. The $< r/c >$ field in the instruction indicates which of the two 1D-arrays is active. The *R_row_enb* and *R_col_enb* registers indicate active processors and operands during execution. Similar to 2D-array instructions, the values for quality fields are obtained from registers *R_q_type* and *R_q_amt*.

1D-array Streaming Instructions: In this instruction type, data operands are streamed in from either the 2D-array or streaming memory, and are stored back

in their respective registers after being operated upon in the 1D-array processing elements. The instruction fields are similar to 1D-array reduction instructions.

1D-array Self Operand Instructions: The operand for this instruction type is obtained from the register file of the 1D-array processing elements. These instructions typically execute in a single cycle. The mask register, as in conventional SIMD architectures, allows 1D-array processing elements to execute conditional (if-then-else) statements. Instructions to conditionally set/reset the mask register also belong to this category.

In summary, the proposed ISA is tailored to efficiently execute a wide range of intrinsically resilient applications and has provisions for expressing application resilience to hardware using the instruction-level quality fields.

3.4.2 QUORA Micro-architecture

The architecture of QUORA was primarily driven by studying the characteristics of numerous resilient applications. It also features several micro-architectural design choices that enhance the energy savings achieved through approximate computing. The block diagram of the QUORA architecture is shown in Figure 3.5. This section describes in detail the various architectural features and components of QUORA. An in-depth description of the different types of processing elements and their interconnection patterns is also provided.

Processing Element Hierarchy

QUORA contains three types of processing elements (PEs) that differ widely in several aspects including their functional complexity, programmability, number and ability to scale under approximate operation. A description of the different types of PEs is provided below.

Approximate Processing Elements: The approximate processing elements (APEs) are the simplest and most in number among the three types of PEs. As shown in Fig-

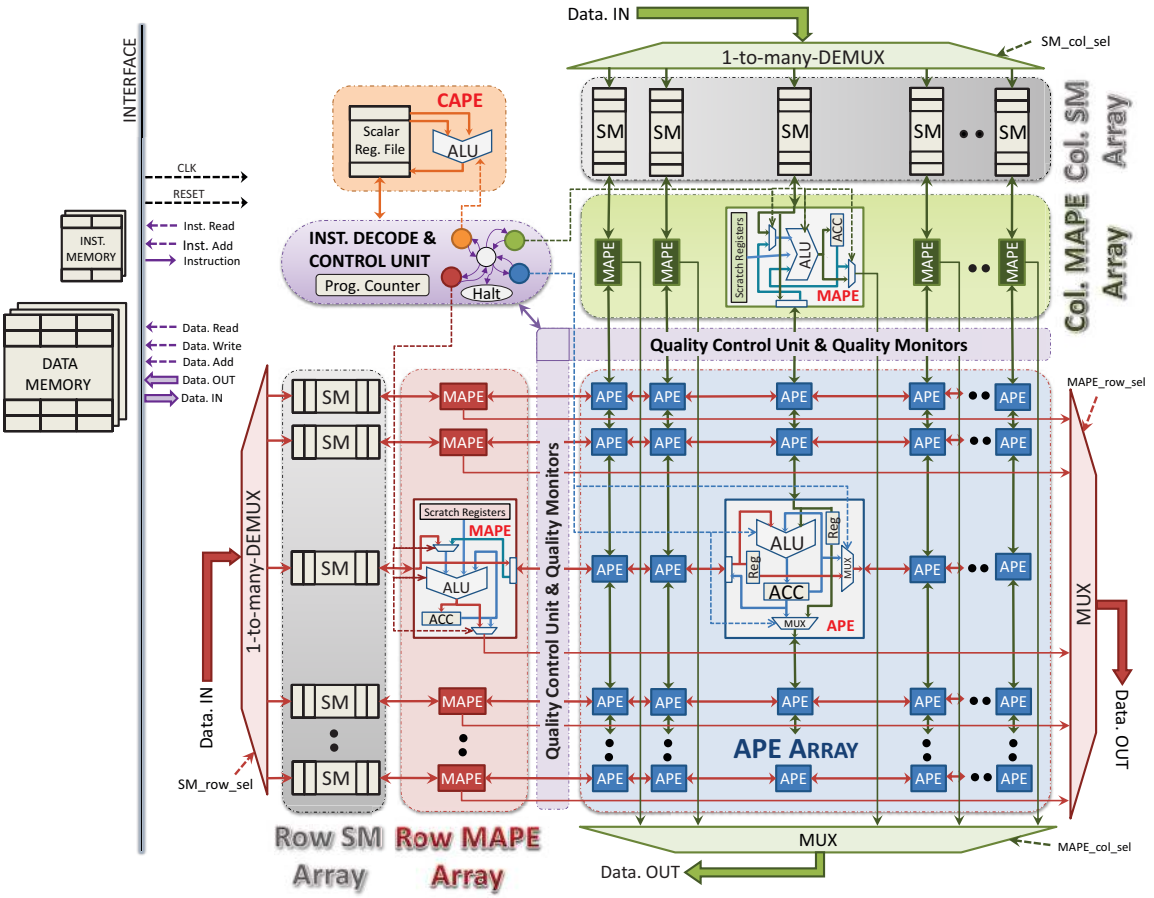


Fig. 3.5.: QUORA micro-architecture

ure 3.5, the APEs are arranged in a 2-dimensional array and are interconnected in a systolic fashion with every APE connected only to its nearest neighbours. Figure 3.6 shows the block diagram of an APE. Each APE contains an accumulator register and takes 2 operands as inputs. The input operands are processed through a 2-level datapath and the results are accumulated with the value stored in the accumulator register. The functionality of each level in the APE datapath can be selected using control signals.

In a typical instruction, an APE operates over multiple cycles with new input operands fed in every cycle from neighboring APEs to the left and top of it. The APE elements in the left and top borders of the 2-D array take inputs from streaming

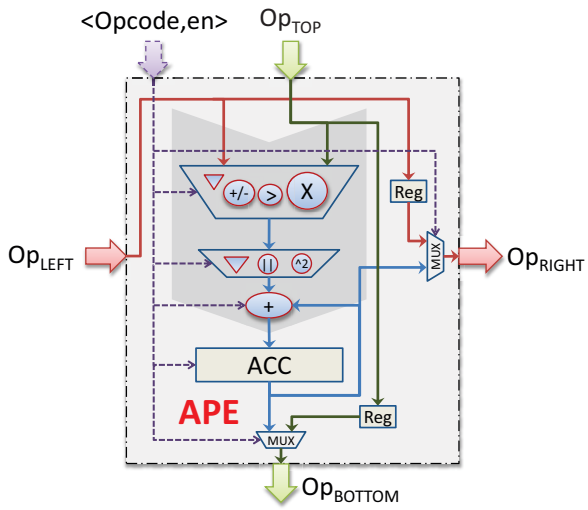


Fig. 3.6.: Approximate processing element

memory elements located along the borders. The input operands are registered in each APE and are propagated to its right and bottom neighbours. The accumulator contains the output at the end of execution. The accumulators present in a given row or column of the array are also connected with each other. Thus, the accumulator outputs can be scanned out to memory or shifted to neighboring APEs, in either row or column fashion.

The APE elements are optimized to carry out vector-vector reduction operations such as Dot product, Euclidean distance *etc.*, which are very common in resilient applications. Since most APE interconnects are fixed, they contain minimal control logic and over 90% of their energy consumption is contributed by the execution units. Note that the control logic is evaluated once at the start of the operation, while the execution units operate over a large number of cycles. Hence among the PEs, the APEs' energy consumption scales the best under approximation.

Mixed Accuracy Processing Elements: QUORA contains mixed accuracy processing elements (MAPEs) arranged along the left and top borders of the APE array, as shown in Figure 3.5. The MAPE elements are divided into two sets *viz.* row MAPEs, present on the left border, and column MAPEs, along the top border. The

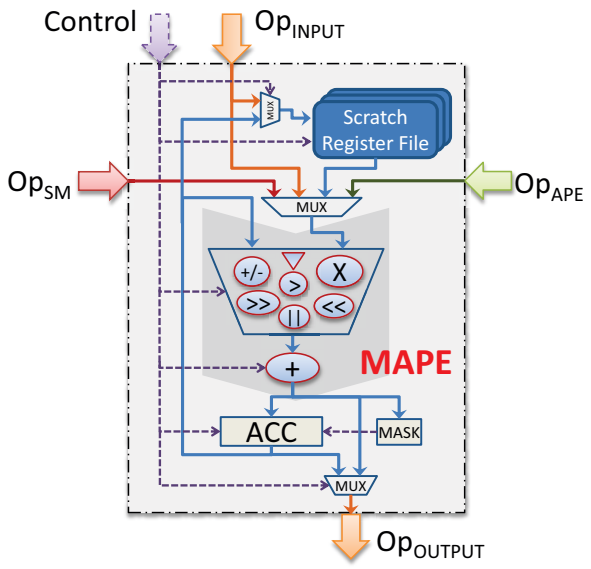


Fig. 3.7.: Mixed accuracy processing element

MAPEs in each set operate together in an SIMD fashion. Figure 3.7 shows the block diagram of a MAPE. It contains an accumulator register and along with a small register file. The execution unit is more sophisticated compared to APE and always takes the accumulator register as one of its operands. The second operand could be obtained from the accumulator register of the APE element located on same row/column border as the MAPE element. However, the operand can also be fetched from the streaming memory element located beside the MAPE or from its scratch register file. The output of the execution unit is always stored back to the accumulator.

The MAPEs operate in 3 different modes: (i) Reduction mode, (ii) Streaming mode, and (iii) Self-operand mode. Both reduction and streaming modes contain multi-cycle operations where the data operands to MAPEs are either scanned out from the APE array or from the streaming memory. In the case of reduction mode, the output is reduced within the accumulator of MAPE, whereas in the streaming mode, the result is stored back to the APE or SM that supplied the data operand in a cyclical fashion. The self-operand mode is a single cycle operation in which the data operand is fetched internally from the register file of the MAPE.

The MAPEs are designed execute a mix of arithmetic and control instructions. To facilitate evaluation of conditionals (if-then-else statements), the MAPE elements contain a mask register which can be set/reset based on the output of the execution unit. A MAPE operates only if the mask register is unset. The MAPE accumulator registers can be scanned out to memory in a manner similar to the APE accumulator registers.

The applications of interest typically produce a large amount of intermediate data on which complex reduction operations such as finding min/max, kernel functions like tanh, exponential *etc.* are performed. MAPEs are tailored to perform such operations. More than 70% of the energy consumed by MAPEs is contributed by the execution unit and can be scaled under approximate operation. Thus, while MAPEs are more complex and hence less scalable compared to APEs, they still offer significant opportunities for approximate computing.

Completely Accurate Processing Element: Quora contains one completely accurate processing element (CAPE), which is similar to a conventional scalar microprocessor. It contains a register file from which operands are fetched and operated upon in its execution unit. As the name implies, the operations performed by CAPE are done in an completely accurate manner. CAPE is typically used to execute instructions that relate to the control flow of the program. This includes computations for loop control, address computation, pointer arithmetic, *etc.*

In summary, as illustrated in Figure 3.8, Quora contains a 3-tiered hierarchy of processing elements that vary in functionality and energy scalability under approximations. For a given array configuration of n rows and m columns, the architecture contains $n*m$ APEs, $n+m$ MAPEs and 1 CAPE. Hence, although individual CAPE and MAPE elements are more complex, the energy consumption of Quora is dominated by the APEs followed by MAPEs and the CAPE. The above design choices, while allowing efficient execution of different applications, also enable significant energy benefits to be derived from approximate computing.

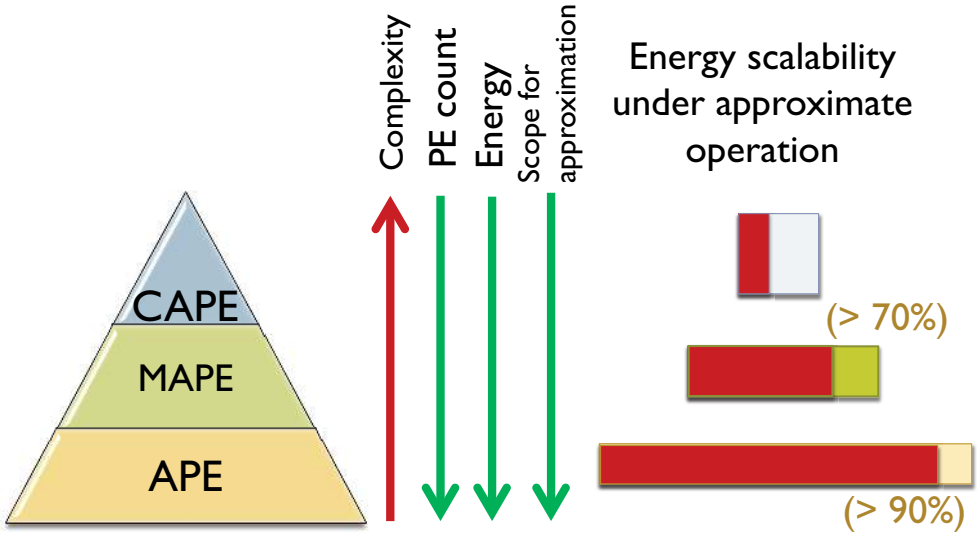


Fig. 3.8.: Comparison of processing elements in QUORA

Streaming Memory Elements

The streaming memory (SM) elements are First-in-first-out (FIFO) circular buffers located along the left and top borders of the APE array, as shown in Figure 3.5. Data operands are loaded into one or more of these SM elements from the data memory, and subsequently consumed by the APEs and MAPEs. As seen from Figure 3.5, each SM element is connected to exactly one APE and MAPE element located beside it.

Quality Control Unit and Quality Monitors

The quality control unit interprets the quality fields present in each instruction and translates them, based on the instruction type, into the hardware accuracy knobs for the APE and MAPE processing elements. Any approximate design technique may be employed in the PEs, provided that suitable quality translation is performed in the quality control unit. In QUORA, precision scaling of input data operands is employed to achieve quality configurability in the PEs. Different flavors of precision scaling mechanisms are proposed for this purpose. Using precision scaling allows the quality control units to be shared across APEs in a row/column. Hence, they are located

along the borders of the APE array as shown in Figure 3.5. The quality monitors that estimate the error accrued during instruction execution are also shared among PEs and are located along with the quality control units. Detailed descriptions of various precision scaling techniques and their associated benefits and overheads are provided in Section 3.5.

Instruction Fetch, Decode and Control Units

Quora contains a program counter that points to the address from which the next instruction is fetched. The instruction decode unit identifies the type of instruction to be executed on the APE, MAPE and CAPE elements. The control unit initiates the instruction execution by asserting appropriate control signals to the processing elements and streaming memories.

3.5 Micro-architectural Mechanisms for Quality Scaling

In order to facilitate quality-configurable execution on the processing elements, Quora is designed with hardware mechanisms that provide trade-off between computational accuracy and energy efficiency. Quora employs precision scaling, where the bit width (precision) of input operands used by the processing elements (PEs) are modulated during instruction execution. In this section, we describe the different flavours of precision scaling used in Quora, along with their benefits and overheads. The design of the *quality control unit* that translates instruction quality specifications to the hardware precision scaling knob, and *quality monitors* that provide an estimate of actual error in the execution are also outlined.

3.5.1 Precision Scaling

Precision scaling is a commonly used approximate design technique, in which some pre-determined number of least significant bits (LSB) in the data operands to the PEs

are ignored during computation. This can be exploited for power savings by clock gating or power gating parts of the datapath that are no longer necessary for reduced precision operation. We leverage the fact that quality programmable instructions in QUORA often operate on data vectors over multiple cycles, and propose several variations of precision scaling to achieve improved energy-quality trade-offs. These mechanisms are described in the sections below.

Up/down Precision Scaling

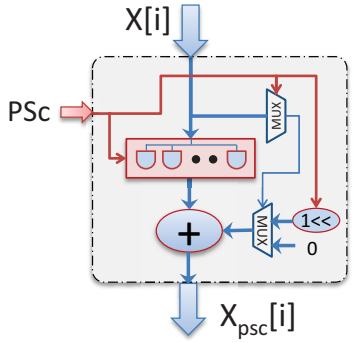


Fig. 3.9.: Up/down precision scaling

Figure 3.9 shows the block diagram of the up/down precision scaling unit. It takes a data operand (X) and the number of bits (PSc) by which it should be precision scaled as its inputs. The precision scaled version of the operand (X_{psc}) is produced at the output. The up/down precision scaling mechanism is similar to round-off in floating point notation. The LSB PSc bits of the input operand are first set to zero, as in conventional truncation. However, if the value of the truncated bits is greater than or equal to 2^{PSc-1}, then the input is rounded up by adding 2^{PSc} to the bit-truncated version. For example, for a PSc value of 3, if the last 3 bits of the input operand represents a value greater than or equal to 4, then 8 is added to the truncated input value. The condition for rounding up *vs.* down can easily be detected in hardware by checking if the $(PSc-1)$th bit of the input is equal to 1.

Precision Scaling with Error Monitoring

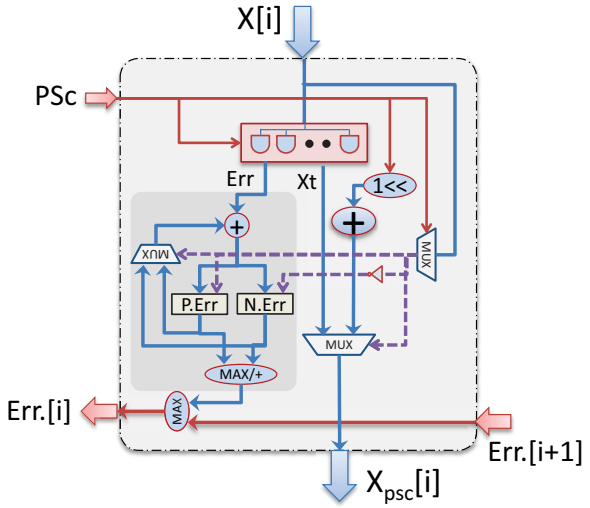


Fig. 3.10.: Precision scaling with error monitoring

The error incurred in scaling precision should be monitored in order to evaluate its impact on overall instruction output quality. The block diagram of the precision scaling unit with error monitoring is shown in Figure 3.10. Up/down precision scaling introduces errors that are either positive or negative, based on the direction in which the value is rounded. Hence, the error monitoring circuit contains separate registers, $P.Err$ and $N.Err$, to track positive and negative errors respectively. Typically, quality programmable instructions in QUORA operate on data vectors, whose individual elements are precision scaled in successive execution cycles. Hence the error monitoring circuit contains the provision to accumulate error over multiple cycles by accordingly updating the error registers. The $P.Err$ and $N.Err$ registers are used to estimate the overall error at the output of an instruction, as detailed in section 3.5.4.

Precision Scaling with Error Compensation

The up/down precision scaling, by virtue of introducing errors in either direction, inherently possesses the ability to compensate for errors that accumulate over mul-

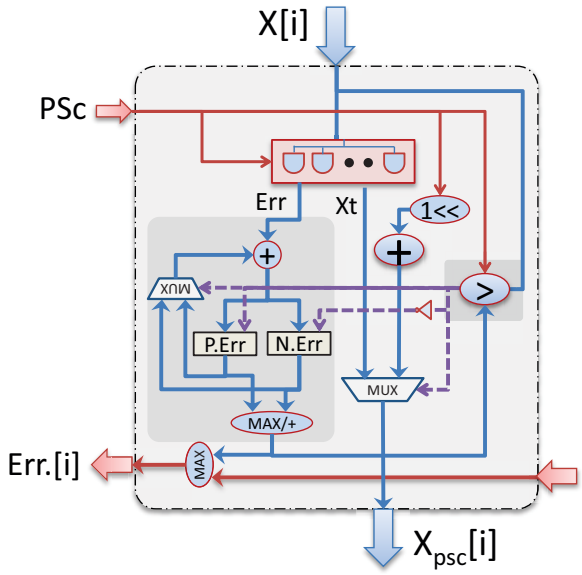


Fig. 3.11.: Precision scaling with error compensation

tiple cycles. We enhance the error compensating nature by dynamically varying the threshold at which the values are rounded up/down in the precision scaling unit. The block diagram for precision scaling with enhanced error compensation is shown in Figure 3.11. The logic used to decide the rounding direction contains a comparator whose threshold is varied based on the values in the error register. At the start of instruction execution, both error registers are reset to zero and the threshold is set to 2^{PSc-1}. However, during the course of execution, if the positive or negative error is greater than twice the other, the comparator threshold is decreased or increased respectively. Increasing the threshold causes more values to be rounded down, thereby introducing more positive errors, and vice versa. This enhanced error compensation significantly improves the energy-quality trade-off offered by QUORA.

Dynamic Precision Scaling

In the previous mechanisms, the number of bits (PSc) by which the operands are precision scaled is set once at the start instruction execution. We present a

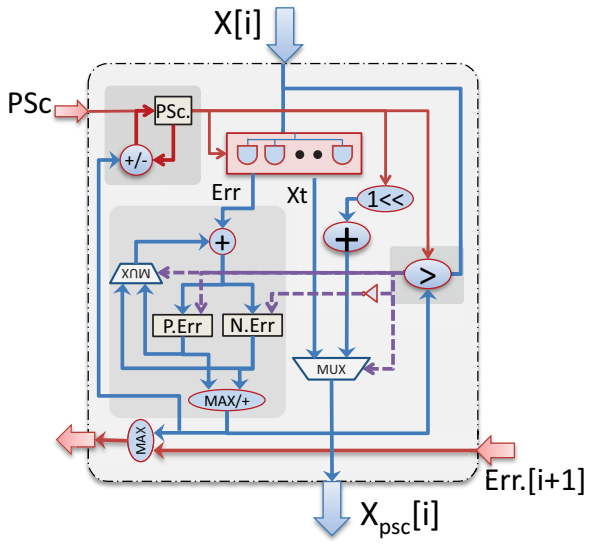


Fig. 3.12.: Dynamic precision scaling

final variant, dynamic precision scaling, where PSc is changed dynamically during instruction execution based on the actual value of error. Figure 3.12 shows the block diagram of a dynamic precision scaling unit. The error accrued in the error registers are checked periodically and a new PSc value is determined based on the error.

3.5.2 Array Level Organization of Precision Scaling Units

This section describes how the precision scaling units (PScU) are integrated into QUORA. The quality programmable instructions execute on the APE/MAPE elements, whose data operands are fetched from either the streaming memory elements or the accumulator registers present in the APE array. Hence, the precision scaling units are placed along the left and top borders of the array between the APEs and MAPEs, as shown in Figure 3.13. In case of instructions that operate on the APE array, data from the streaming elements are precision scaled before being fed into the APEs. Similarly, when the accumulator registers are scanned out for MAPE operation, precision scaling is employed on those operands.

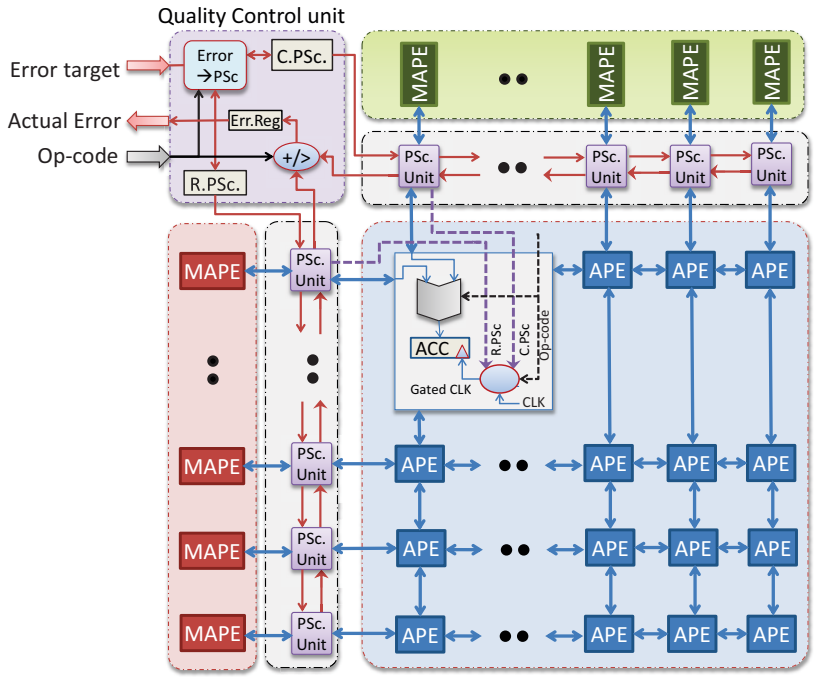


Fig. 3.13.: Array level organization of precision scaling units

Quality Control Unit

Figure 3.13 also shows the block diagram of the quality control unit. The quality control unit maps the target quality specified in the instruction, based on its op-code, to the precision scaling values for PScUs located along the rows ($R.PSc$) and columns ($C.PSc$).

Quality Monitors

The logic for quality monitoring is distributed among individual precision scaling units and the quality control unit. The positive and negative error registers in the precision scaling units contain an estimate of error introduced in their respective data vectors. These error registers are scanned along row and column precision scaling units and used by the quality control unit to obtain the overall error at the output

of the instruction. The estimated error is stored in the error register, *Err.Reg*, of QUORA.

The quality translation and error estimation procedures are instruction dependent and are described for various instructions and quality types in section 3.5.4.

3.5.3 Precision Scaling: Impact on Energy

Precision scaling of operands has significant impact on the energy consumed by QUORA. A qualitative evaluation of the various benefits and overheads are described in this section.

Benefits

The energy benefits due to precision scaling stem from three direct sources.

- **Reduction in Switching Activity** When precision scaled, the LSB bits of the data operands are set to zero. This results in reduction in the switching activity in the PEs, as the logic slices corresponding to the precision scaled bits are dormant. Hence, the dynamic power consumed by the PEs are inherently reduced under precision scaled operation.

- **Clock gating of registers**: As the precision of the computations performed in the PEs is scaled, bit slices of the registers can be clock gated to further enhance their energy efficiency. As shown in Figure 3.13, signals for clock gating are generated based on the *PSc* values in the quality control unit and fed to the APEs and MAPEs.

- **Voltage scaling**: Precision scaling also leads to reduction in circuit delay due to reduction in the critical path of the design. This can be used to benefit energy consumption, by reducing the supply voltage to the APE and MAPE processing elements. The quality programmable instructions in QUORA execute over multiple (tens to hundreds) clock cycles and hence supply voltage modulation

is feasible. Note that supply voltage reduction is only employed to exploit the timing slack created by precision scaling, and no timing errors are introduced in the PEs.

In addition to the above direct benefits, precision scaling provides an indirect energy benefit in the execution of subsequent instructions. Precision scaling the input operands of an instruction naturally results in its outputs being computed with a lower precision. Thus, subsequent instructions that consume the output value of the current instruction can also be precision scaled without any additional degradation in their output quality. The hardware is equipped to store the precision of accumulator registers in APEs/MAPEs, which are then selectively used to scale the precision of future instructions that operate on these values.

Overheads

In Quora, the key hardware overheads of precision scaling are the precision scaling units, which are located only along the top and left borders of the APE array, and the quality control logic, which is global to the processor. The precision scaling units are significantly smaller than the processing elements and their cost is further amortized as they are shared among APEs in a given row or column. The APEs are augmented with logic to enable clock gating of registers. In our implementation, the hardware overheads for precision scaling contribute $< 1\%$ of the total energy consumed by other components.

3.5.4 Quality Translation and Error Estimation

Quality translation entails determining the row ($R.PSc$) and column ($C.PSc$) precision scaling values based on the type and amount of error specified in the quality fields of the instruction. The error monitoring registers present in the precision scaling units keep track of the error introduced in the input operands of the instruction. At the end of instruction execution, this error at the inputs is translated into error at

the instruction output and stored in the error register ($Err.Reg$) of QUORA. The quality translation and error estimation procedures are instruction specific and are shown for a representative subset of quality programmable instructions in Table 3.2.

Table 3.2.: Quality translation and error estimation for quality programmable instructions

Inst. Type	Inst.	Maximum Error Magnitude	Average Error Magnitude
2D Array Inst.	qpMAC	**C.PSc** $= 1 + lg_2 \left\lfloor \frac{MaxErr}{length * 2^{BW}} \right\rfloor$ **R.PSc** $= 0$ **E. Reg** $= 2^{BW} * \max(\lvert P.Err \rvert, \lvert N.Err \rvert)$	**C.PSc** $= 2 + lg_2 \left\lfloor \frac{AveErr}{length * 2^{BW}} \right\rfloor$ **R.PSc** $= 0$ **E. Reg** $= 2^{BW-1} * \max(\lvert P.Err \rvert, \lvert N.Err \rvert)$
	qpSMAC (Signed MAC)	**C.PSc** $= 1 + lg_2 \left\lfloor \frac{MaxErr}{length * 2^{BW}} \right\rfloor$ **R.PSc** $= 0$ **E. Reg** $= 2^{BW} * (\lvert P.Err \rvert + \lvert N.Err \rvert)$	**C.PSc** $= 2 + lg_2 \left\lfloor \frac{AveErr}{length * 2^{BW}} \right\rfloor$ **R.PSc** $= 0$ **E. Reg** $= 2^{BW-1} * (\lvert P.Err \rvert + \lvert N.Err \rvert)$
	qpMOD2	**R.PSc** $=$ **C.PSc** $= lg_2 \left\lfloor \frac{MaxErr}{length * 2^{BW}} - 1 \right\rfloor - 1$ **E. Reg** $= 2^{BW+1} * \Delta(1+\Delta)$ *where* $\Delta = \max(PErr_R + NErr_C, NErr_R + PErr_C)$	**R.PSc** $=$ **C.PSc** $= lg_2 \left\lfloor \frac{2 * AveErr}{length * 2^{BW}} - 1 \right\rfloor - 1$ **E. Reg** $= 2^{BW} * \Delta(1+\Delta)$ *where* $\Delta = \max(PErr_R + NErr_C, NErr_R + PErr_C)$
1D Array Inst.	qpACC	**R.PSc /C.PSc** $= 1 + lg_2 \left\lfloor \frac{MaxErr}{length} \right\rfloor$ **E. Reg** $= \lvert P.Err \rvert - \lvert N.Err \rvert$	**R.PSc /C.PSc** $= 2 + lg_2 \left\lfloor \frac{AveErr}{length} \right\rfloor$ **E. Reg** $= \lvert P.Err \rvert - \lvert N.Err \rvert$
	qpMUL	**R.PSc /C.PSc** $= 1 + lg_2 \lfloor MaxErr \rfloor$ **E. Reg** $= 2^{BW} * (\lvert P.Err \rvert + \lvert N.Err \rvert)$	**R.PSc /C.PSc** $= 1 + lg_2 \lfloor AveErr \rfloor$ **E. Reg** $= 2^{BW-1} * (\lvert P.Err \rvert + \lvert N.Err \rvert)$

We describe the procedure in detail for the multiply-and-accumulate instruction ($qpMAC$: row 1 of Table 3.2) using the maximum error magnitude quality metric. In the case of $qpMAC$, only one of the input operands (column operand in QUORA) is precision scaled. Equation 3.5a shows the analytical expression for error magnitude at the instruction output when one of the operands is precision scaled. In Equation 3.5a, A_i and B_i are the input operands, and Δ_i is the error due to precision scaling B_i.

$$ErrMag = |O_{acc} - O_{approx}| = \Big| \sum_{i=1}^{n}(A_i B_i) - \sum_{i=1}^{n}(A_i(B_i + \Delta_i) \Big| = \sum_{i=1}^{n}(A_i \Delta_i) \quad (3.5a)$$

$$\begin{aligned} MaxErr \geq \sum_{i=1}^{n}(A_i \Delta_i) \geq n A_{max} \Delta_{max} \geq n 2^{BW} 2^{C.PSc-1} \\ \implies C.PSc = 1 + \log_2 \Big\lfloor \frac{MaxErr}{n 2^{BW}} \Big\rfloor \end{aligned} \quad (3.5b)$$

We need to ensure that this error ($ErrMag$) is guaranteed to be lower than the value, $MaxErr$, specified in the instruction quality amount field. Equation 3.5b also shows the steps in calculating $C.PSc$ under this condition. BW is the bit width of the operation. As we can see from Equation 3.5b, the quality translation process is conservative in-order to ensure the quality bounds are guaranteed for all possible inputs.

The total positive and negative error in the input operands are available in the $P.Err$ and $N.Err$ registers of the precision scaling units. In the worst case for *qpMAC*, the error estimate is the maximum of $P.Err$ and $N.Err$ multiplied by the maximum possible value of the other operand.

In summary, the precision scaling mechanisms enable quality-configurable execution of instructions in Quora and the quality control unit regulates these mechanisms based on the instruction-level quality specifications.

3.6 Evaluation Methodology

In-order to demonstrate the energy benefits of quality programmability, we realized a hardware implementation of Quora and performed a wide range of experiments on it using a suite of popular applications from the recognition, mining and synthesis application domains. This section describes our design flow, experimental methodology and the benchmarks used in evaluation.

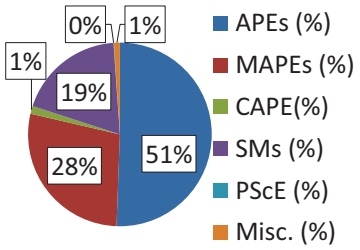
3.6.1 RTL Implementation

QUORA was implemented at the register-transfer level (RTL) using Verilog HDL and synthesized to the IBM 45nm technology library using Synopsys Design Compiler [34]. The micro-architectural parameters and implementation metrics are shown in Table 3.3.

Table 3.3.: QUORA: Micro-architectural parameters and implementation metrics

Micro-architectural Parameters	Value
Array Dimensions	16 X 16
No. of PEs (APEs + MAPEs+ CAPE)	289 (256 + 32 + 1)
Size of Register File – CAPE / MAPE	32 / 8
No. of SM elements	32
Depth of SM elements	64
Operating Frequency	250 MHz

Metric	Value
Feature Size	45nm
Area	2.6 mm^2
Power	367.8 mW
Gate Count	502042



Synopsys Power Compiler [34] was used to estimate power consumption at the gate-level. Table 3.3 also shows the power breakdown among different components of QUORA. The APE and MAPE elements contribute 79% of the total power, providing ample scope for obtaining energy benefits through approximate computing.

3.6.2 Application Benchmarks

Table 3.4 lists the applications and datasets that were used as benchmarks in our experiments. The application-specific metrics used to evaluate the output quality are also provided.

Table 3.4.: QUORA: List of application benchmarks

Applications	Algorithm	Dataset	Quality Metric
Handwritten Digit Recognition (SVM-MNIST)	Support Vector Machines	MNIST	Percentage Classification Accuracy
Object Recognition (SVM-NORB)	Support Vector Machines	NORB	
Digit Classification (CNN)	Convolutional Neural Networks	MNIST	
Eye Detection (GLVQ)	Generalized Learning vector Quantization	Image set from NEC labs.	
Optical Character Recognition (k-NN)	K-nearest Neighbors	OCR digits	
Census Data Analysis (ANN)	Artificial Neural Networks	Adult	
Document Search (SSI)	Semantic Search Index	Subset of Wikipedia	No. correct in top 25 results
Image Segmentation (K-Means-Seg)	K-means Clustering	Berkeley dataset	Mean distance of clustered points from respective centroids
Optical Character Clustering (K-Means-OCR)	K-means Clustering	OCR digits	

The applications were ported manually to the QUORA ISA and the quality fields of quality programmable instructions were set empirically using a greedy auto-tuning procedure with iterative quality profiling as described below. First, the quality fields in all quality programmable instructions in the assembly program are set as fully-accurate. Next, the program is energy-profiled to identify the most energy consuming instruction and its accuracy relaxed in steps of 2.5%. The application is executed on an instruction-level simulator to evaluate application-level quality. This process is

repeated until the application-level quality constraint is violated to obtain to the quality programmable assembly code.

3.6.3 Energy and Quality Measurements

Gate-level Energy Estimation for Micro-Benchmarks:

Since energy estimation of the entire application at the gate-level incurs prohibitively large run-times, we used individual instructions in the ISA as micro-benchmarks and evaluated their energy consumption. Table 3.5 shows the number of execution cycles and energy for a subset of instructions. In the case of quality programmable instructions, the energy consumed varies based on the desired accuracy level and is shown as an energy range in Table 3.5. Note that these instructions, in addition to executing on APE and MAPE processing elements, extend over a large number of clock cycles compared to scalar instructions, enhancing their total energy consumption. Also, since APEs and MAPEs dominate the energy consumption in QUORA, a significant fraction (¿80%) of the quality programmable instruction energy is scalable under approximate operation.

RTL and Instruction-accurate Simulation

We used *ModelSim* [44] to simulate complete applications at the register-transfer level and obtained the dynamic instruction count of various instructions in each program. We combined the energy consumed by individual instructions with their dynamic count to compute the total energy consumed by QUORA for executing the entire application. Since precision scaling input operands only introduces functional errors during execution, we developed an instruction accurate simulator and used it to obtain the overall application output quality.

Table 3.5.: Execution time and energy of instructions in QUORA's ISA

Instruction	Execution Cycles	Energy (nJ)	Fraction of energy Scalable under Approx. operation
LDRI	3	0.012	--
ADDR	3	0.015	--
LDSM	11	0.093	--
qpMAC	96	43.26 – 13.03	0.981
qpMOD2	96	38.54 – 9.63	0.979
STR	259	2.43	--
qpACC	18	0.98 – 0.76	0.829
qpMIN	18	1.14 – 0.88	0.853
qpADDX	3	0.07 – 0.05	0.885
qpMUL	3	0.17– 0.12	0.953

3.7 Experimental Results

In this section, we present the results of various experiments that demonstrate significant benefits obtained by leveraging quality programmability in the benchmark applications.

3.7.1 Energy Benefits

Figure 3.14 shows the normalized energy consumption at different output quality targets for all applications. The energy is normalized to the base case where all the quality programmable instructions in the application are executed in a fully accurate manner. The energy benefits range between 1.05X-1.7X for virtually no loss ($< 0.5\%$)in output quality and between 1.18X-2.1X when modest loss ($< 2.5\%$) in quality is tolerable. When the quality requirements are further relaxed ($< 7.5\%$), the benefits extend upto 2.5X compared to fully-accurate QUORA baseline. On an aver-

age, leveraging quality programmability results in 1.3X, 1.6X, and 1.85X at different quality levels respectively.

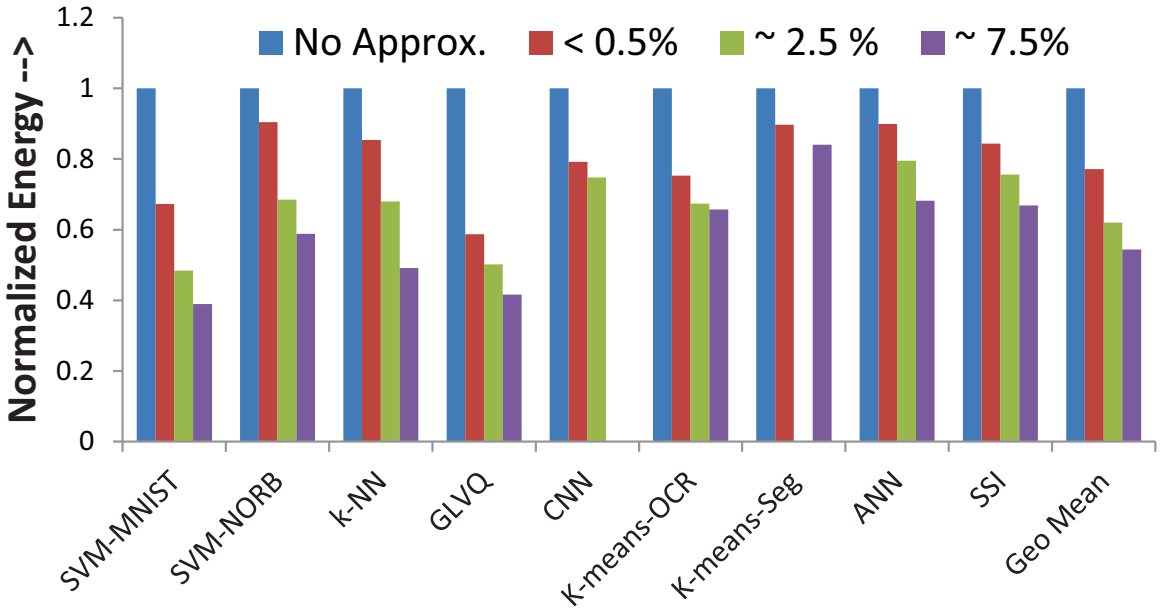


Fig. 3.14.: Energy benefits for different application-level quality constraints

3.7.2 Quality Programmability in Instructions

To demonstrate the need for quality programmability in instructions, we examine the energy and output quality of applications for different instruction level quality bounds. Figure 3.15 shows the plot for 2 applications, viz. hand digit recognition (MNIST dataset) and object classification (NORB dataset), both using the Support Vector Machines (SVM) algorithm. In both cases, significant energy improvements are obtained at lower error bounds for no degradation in application output quality. However, after some point, energy benefits begin to taper off and the degradation in application output quality becomes marked. Moreover, as seen from the graphs, based on the application context and the dataset being processed, the same algorithm exhibits significantly different energy-quality trade-offs. Thus, for a given target output quality, to maximally exploit the benefits of error resilience, we require instructions

to be executed at different accuracy levels. Further, the target quality requirement itself may vary based on the context in which the application output is used. These factors underscore the importance of quality-programmability in instructions.

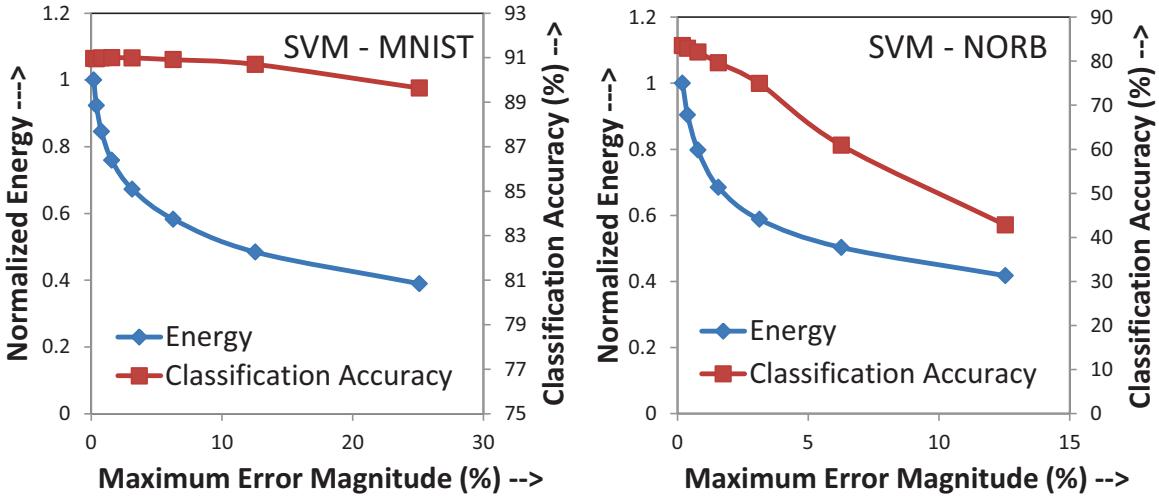


Fig. 3.15.: Energy reduction and application-level quality degradation for different instruction level quality specifications

3.7.3 Energy Contribution of Quality Programmable Instructions

To better understand the scope of benefits that can be obtained by approximate computing in QUORA, we present the contribution of quality programmable instructions to the dynamic instruction count, execution cycles and overall energy for various applications, in Figure 3.16.

On an average, we observe that, when applications are ported to QUORA, less than 2% of the instructions are quality programmable. These instructions typically execute over multiple cycles contributing about 20% of total application runtime. However, these instructions execute on the APE and MAPE elements, which enhance their total energy contribution to more than 75%. This observation is typical of many error resilient applications which contain few dominant kernels that are resilient and consume a significant portion of application energy [4]. Thus, in QUORA, significant

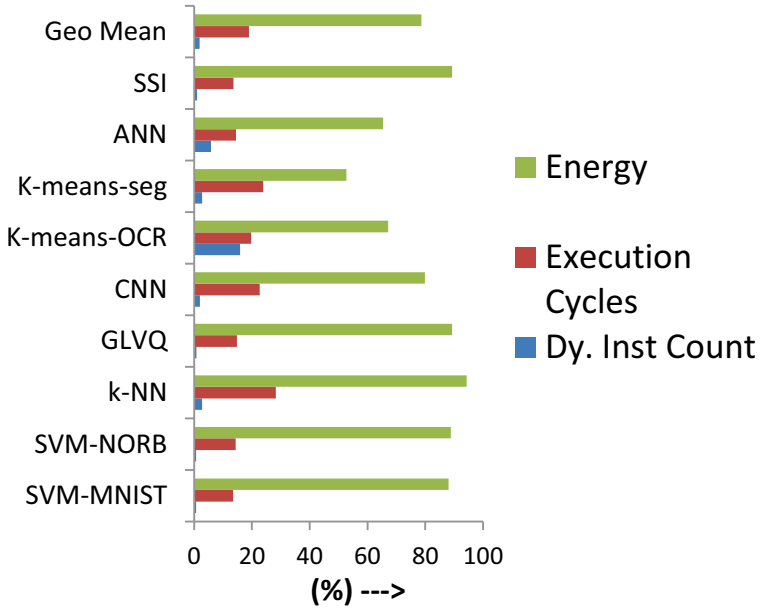


Fig. 3.16.: Contribution of quality programmable instructions to dynamic instruction count, execution cycles and energy

energy benefits can be realized even when pogram uses a small number of quality programmable instructions.

3.7.4 Precision Scaling Mechanisms

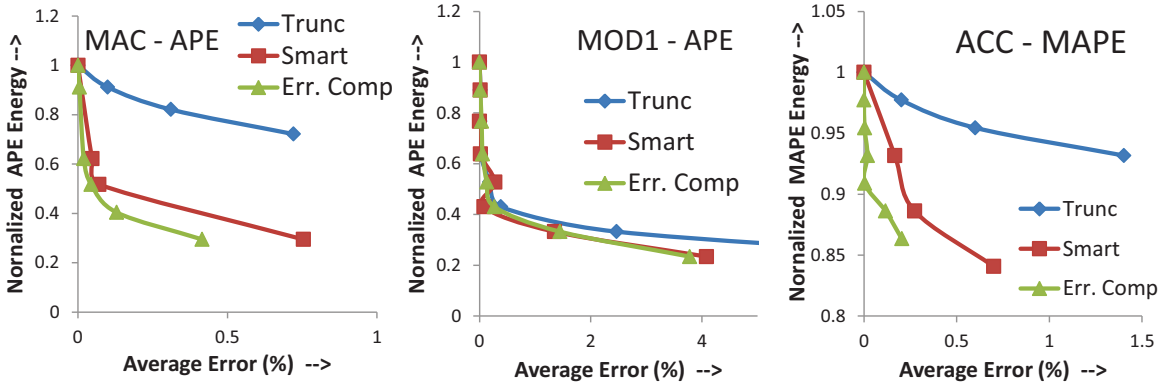


Fig. 3.17.: Energy *vs.* error curves for micro-benchmarks using different precision scaling mechanisms

The impact of employing the proposed precision scaling mechanisms is studied using different micro-benchmarks and the results are presented in form of energy *vs.* error graphs in Figure 3.17. We compare 3 schemes: (i) Truncation, where the LSB bits of inputs operands to the instruction are set to zero, (ii) Up/down precision scaling, where LSB bits are rounded up/down based on a fixed threshold, and (iii) Precision scaling with error compensation, in which the threshold used in up/down precision scaling is varied based on the actual error incurred during execution. The micro-benchmarks were executed on 10000 random input instances. We can observe from Figure 3.17 that precision scaling with enhanced error compensation outperforms the other schemes and provides the best energy-quality trade-off in all 3 cases.

3.7.5 Architectural Exploration

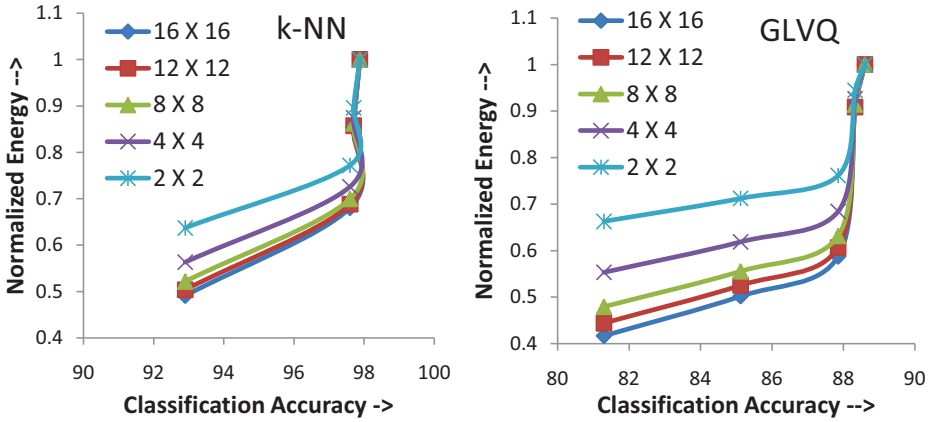


Fig. 3.18.: Energy *vs.* quality curves for varying array dimensions

To study the impact of micro-architectural parameters on energy benefits, we perform an architectural exploration by varying QUORA's array dimensions. Figure 3.18 shows the energy *vs.* output quality (classification accuracy) graphs for two applications, k-NN and GLVQ. Exploiting quality programmability yields considerable benefits across all the configurations. However, the benefits decrease as the array dimensions are reduced. This can be attributed to: (i) The fraction of energy consumed

by the control front-ends of the design become significant. The overheads increase from 21% for a 16x16 array to 59% for a 2x2 array. (ii) The number of scalar instructions in the program that determine its control flow (such as loop iteration count *etc.*) and their contribution to overall application energy increase – 10% for a 16x16 array to 48% for a 2x2 array for the GLVQ application – as we reduce the parallelism in the processor.

3.8 Summary

To broaden the applicability of approximate computing, this chapter proposed *quality programmable processors*, in which the notion of quality is explicitly codified in the HW/SW interface, *i.e.*, the instruction set. The ISA of a quality programmable processor contains instructions associated with quality fields to specify the accuracy level that must be met during their execution. This chapter demonstrated that this ability to control the accuracy of instruction execution greatly enhances the scope of approximate computing, allowing it to be applied to larger parts of programs. The micro-architecture of a quality programmable processor contains hardware mechanisms that translate the instruction-level quality specifications into energy savings. Additionally, it may expose the actual error incurred during the execution of each instruction (which may be less than the specified limit) back to software.

As an embodiment of the above concepts, this chapter presented a quality programmable vector processor design, Quora, comprised of a 3-tiered hierarchy of processing elements. Based on a 289 processing element RTL implementation and gate-level energy evaluation of Quora, this chapter demonstrated that leveraging quality programmability leads to significant improvements in energy efficiency. In summary, quality programmable processors, by taking approximate computing to the realm of programmable processors, achieves a significant milestone towards bringing approximate computing to the mainstream.

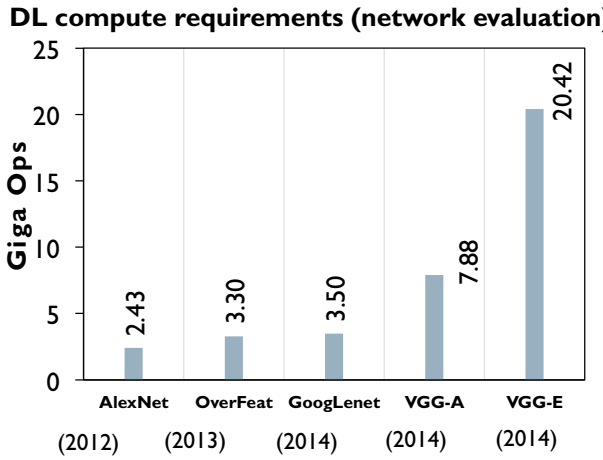
4. ENERGY-EFFICIENT DEEP LEARNING USING APPROXIMATE COMPUTING

4.1 Introduction

The field of neuromorphic computing has garnered significant interest in the past decade due to a confluence of trends from neuroscience, machine learning, semiconductor technology, and high performance computing. An important development within this field has been the advent of large-scale neural networks (NNs) called Deep Learning Networks [39, 45, 46] (DLNs). These biologically inspired algorithms have shown state-of-the-art results on a variety of recognition, classification and inference tasks. Hence, they are deployed in many real world applications such as Google image search [47], Google Now speech recognition [48], and Apple Siri voice recognition [49], among others.

4.1.1 Deep Learning Networks: Computational Challenges

Deep learning networks are highly compute and data intensive due to their large scale and dense connectivity. To understand the computational challenges associated with deep learning, we consider two concrete scenarios that we believe exemplify the most pressing computational challenges: (i) embedding deep learning (network evaluation) in low-power wearable and IoT devices, and (ii) high-speed training of large-scale networks with large data sets. For our analysis, we consider some of the top entries in the ImageNet image recognition competition [46,50–52], which has been dominated by deep networks in recent years.



Overfeat DL network on a Google Glass

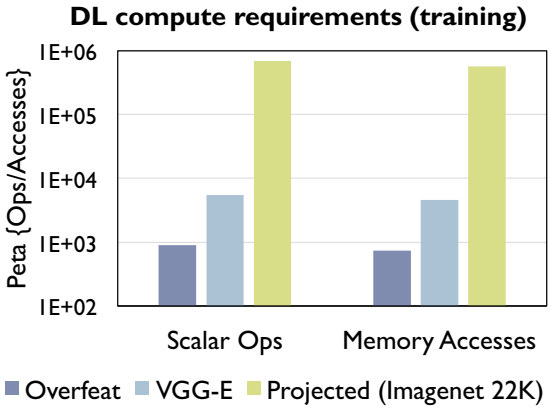
Performance	
OMAP 4430*	1.5 fps (ideal)
Battery Life	
Energy/op (mobile GPU)	5 x 10^{-2} nJ/op
Energy/frame	0.16 J/frame
Time-to-die (2.1WH)	25 min (ideal)

*https://wiki.ubuntu.com/Specs/M/ARMSoCOMAP?action=AttachFile&do=get&target=OMAP_Overview_UDS.pdf

Fig. 4.1.: Computational requirements for embedding deep learning in low-power devices

Deep learning in low-power devices

Consider the problem of processing the video stream captured by a smart glass using a deep learning network that executes on the on-board processor (*i.e.,* the data cannot be uploaded to the cloud due to bandwidth, battery life, or privacy constraints). We note that such applications were explored for the Google Glass [53, 54]. To evaluate the feasibility of this use-case, Figure 2 shows the number of ops required to perform classification using the considered deep networks. Further, using specs from the the Google Glass (OMAP 4430 processor, 2.1WH battery) [55, 56], we compute the frames-per-second that would be achieved assuming the mobile GPU is fully dedicated to evaluation the deep network and runs at peak hardware performance, as well as the battery life if the battery were used only to power the execution of the deep network. As shown from the results in Figure 4.1, there is clearly a need for an order-of-magnitude or higher improvement in processing efficiency to enable real-time recognition in this scenario.



Training VGG-E network on a Xeon Phi server

Performance	
FLOPs/epoch (training)	61.5 Peta-OPs
Training time (Xeon Phi 7120p, peak 2.4TFLOPS)	~27 days (ideal)
Energy	
Training energy (1.5 KW/node)	~972 KWh

Fig. 4.2.: Computational requirements for training deep learning networks in the cloud

Large-scale training in the cloud

If evaluating deep networks is a major challenge at one end of the computing spectrum, building and training these networks is equally challenging at the other end. Let us consider the problem of training a large-scale network for object recognition on the ImageNet data set. Figure 4.2 shows the compute operations and memory accesses required for training current and projected large-scale networks, and the execution time and energy required for training the VGG-E network on an Intel Xeon Phi 7120p processor [57] (again, assuming ideal hardware performance). Clearly, even these ideal numbers indicate that there is a need for large improvements.

4.1.2 Efficiency of DLNs: Prior Research Directions

Previous efforts have explored three major directions for the efficient implementation of NNs, which are outlined in Figure 4.3. The first class of efforts focus on improving the runtime of deep learning networks by parallelizing them on multi- and many-core platforms [61]. The second direction is accelerator based computing, in which custom architectures that are optimized to the computation and communication patterns of NNs are designed. A spectrum of architectures ranging from

application-specific NN designs [58] to programmable neural processors [59, 60] have been proposed.

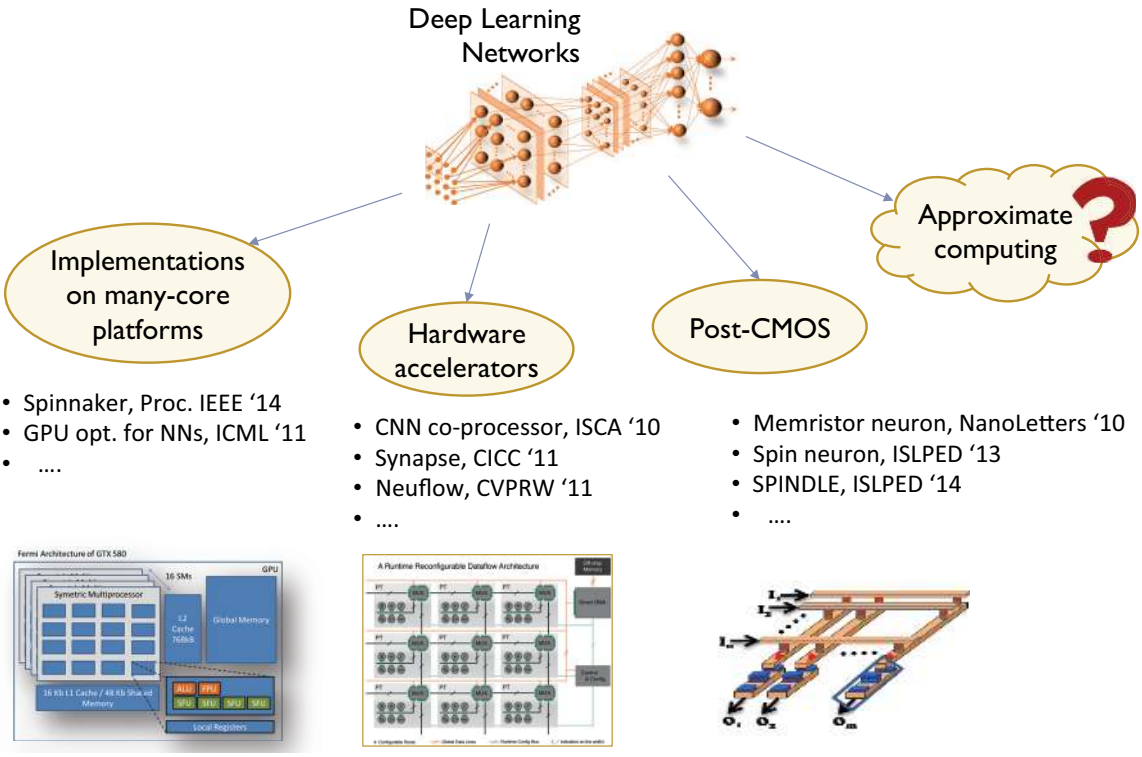


Fig. 4.3.: Research directions to improve deep learning efficiency

The third prominent direction is the use of emerging device technologies to realize the fundamental computations present in NNs. Efficient neuron and synapse implementations based on resistive RAM [62], memristor based crossbar arrays [63], and spintronics [64] have been explored in literature. Complementary to the above efforts, this thesis explores approximate computing as a new dimension to improve the implementation efficiency of deep learning.

4.1.3 Deep Learning ⇔ Approximate Computing

We observe that neural networks, by their very nature, are inherently amenable to approximate computing. First, they are used in applications where less-than-perfect

results are acceptable, and often inevitable. Recent studies [4, 65] have demonstrated that their robustness to noisy, real-world input data also enables them to remain resilient to inexactness or errors in a large fraction of their computations. The resilience to errors is in fact enhanced by the nature of computations performed within a neuron itself. Each neuron in the network evaluates a weighted sum of its inputs, followed by a saturating (or thresholding) non-linear activation function (*e.g.,* tanh, sigmoid). Errors in the positive and negative directions compensate for each other during weighted summation and any residual errors are attenuated by the activation function. Hence, by introducing imperfections in a disciplined manner, the energy consumption of neural networks can be significantly reduced without sacrificing their quality of output.

A key question in approximate computing is which computations to approximate, and by how much. The judicious selection of approximations is critical to maximizing the benefits from approximate computing while ensuring minimal degradation in output quality. For this purpose, it is necessary to determine the impact of approximating various internal computations on the eventual application output quality. In the context of neuromorphic systems, we address this challenge by leveraging backpropagation, an operation that is widely utilized for NN training. We observe that backpropagation provides a measure of the sensitivity of the NN outputs to each neuron in the network; thereby, it can be utilized to identify neurons that are likely to be more resilient to approximations.

The process of training provides a further opportunity to maximize the benefits of approximate computing in NNs. Training is an inherently error-healing process, since it modulates the weights associated with each neuron in the NN such that the error at the network outputs is minimized. Therefore, we suggest that training can also be used to compensate for approximations. Further, this synergy can be exploited in an iterative approximate-and-retrain loop to enhance the benefits of approximate computing.

Based on the above insights, we propose a method to construct ApproXimate Neural Networks (AxNNs) that consists of three key steps. First, it utilizes backpropagation to *characterize* the importance of each neuron in the NN and identify those that impact output quality the least. Next, the AxNN is created by selectively replacing less significant neurons in the network with *approximate* versions that are more energy-efficient. Towards this end, we utilize *precision scaling*, a popular approximate design technique, and modulate the precisions of the inputs and the weights of the neurons to realize versions with different accuracy *vs.* energy trade-offs. Once the approximate NN is formed, we adapt the weights of the neurons in the approximated network by *incrementally retraining* them. Since training is a naturally error-healing process, this allows us to reclaim a significant portion of the quality ceded by approximations. For a given output quality, retraining may create further opportunities to approximate the NN, resulting in increased energy benefits. We develop an automatic design methodology to generate AxNNs by iterating the aforementioned characterize, approximate and retrain steps in a quality-constrained loop.

Another contribution of our work is the design of a quality-configurable Neuromorphic Processing Engine (QCNPE), which provides a programmable hardware platform for efficiently executing AxNNs with arbitrary topologies, weights, and degrees of approximation. QCNPE features a 2D array of Neural Computation Units (NCUs) and a 1D array of Activation Function Units (AFUs) that together enable the efficient execution of neural networks. We equip the NCUs and AFUs with hardware mechanisms based on precision scaling to effectively translate the reduced precision of neurons into energy benefits at run-time.

In summary, the key contributions of AxNN are:

- We propose a new avenue for energy efficiency in neuromorphic systems by using approximate computing. We propose the concept of ApproXimate Neural Networks (AxNNs) that leverage backpropagation to maximize the energy benefits from approximate computing, while utilizing the inherent healing nature of the training process to minimize their impact on output quality.

- Embodying the above design principle, we develop a systematic methodology, which can automatically generate AxNNs for any given neural network. The methodology is independent of the NN topology, network parameters and the training dataset.

- We design a programmable and quality-configurable Neuromorphic Processing Engine (QCNPE) that can be used to efficiently execute AxNNs.

- We construct approximate versions of 6 popular large-scale NN applications using the proposed AxNN design methodology and execute them on two different platforms – QCNPE and commodity Intel Xeon server – to demonstrate significant improvements in energy for negligible loss in output quality.

The rest of the chapter is organized as follows. Section 4.2 provides relevant background on NNs. Section 4.3 outlines the proposed AxNN design methodology. Section 4.4 details the architecture of QCNPE. Section 4.5 describes the experimental methodology and the NN applications used. Section 4.6 presents the results and Section 4.7 concludes the paper.

4.2 Neural Nets: Preliminaries

Neural networks can be broadly described as systems that functionally abstract the computational behavior of the human brain. The fundamental computation unit of NNs is called a *neuron*, which is densely interconnected with several others to constitute a neural network. Each neuron in the network, as shown in Figure 4.4(a), computes a weighted sum of all its inputs, followed by a non-linear activation function on the weighted sum to produce the output.

While the proposed approach can be applied to various classes of NNs, in our discussions we consider the most prevalent form, *viz. feedforward NNs*, wherein the neurons are connected to form an acyclic network, as illustrated in Figure 4.4(b). The operation of NNs typically involves 2 phases *viz.* training and testing. In the training

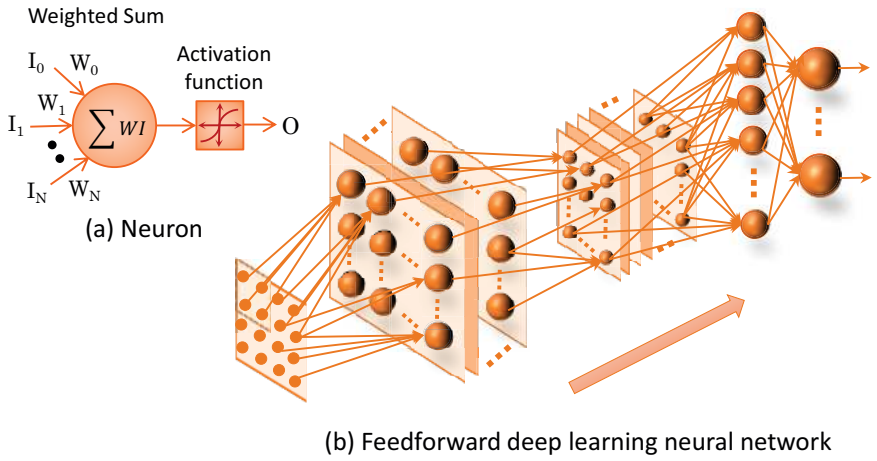


Fig. 4.4.: Neural network preliminaries

phase, the parameters of the NN (weights of each neuron) are identified based on the training dataset. Once the NN is trained, it enters the testing or evaluation phase, in which it is used to perform the desired application. A brief description of the steps involved in testing and training are provided below.

Evaluating NNs—Forward Propagation:

Forward propagation, used widely in both testing and training, is the process of evaluating the outputs of the NN. In forward propagation, the inputs are fed to the neurons in the first layer, where they are processed and propagated to the neurons in the next layer. This process is repeated at all the network layers and the NN outputs are eventually computed.

Training NNs—Backpropagation:

The training process iterates over a dataset of training instances, pre-labeled with golden outputs for the NN, to identify the values of network parameters that maximize the application output quality. The network parameters are typically initialized randomly and are successively refined in each iteration as described below. First, the

NN is evaluated for a random training instance using forward propagation, and the error at the network output (with reference to the golden output) is computed. Next, a key step called *backpropagation* is invoked, which redistributes the error at the NN output backward in the network, all the way to its inputs. Thus, backpropagation quantifies the error contributed by each neuron in the network towards the global network error. Knowing the respective error contributions, the network parameters associated with each neuron are modulated such that the error at its output is reduced. Mathematically, the parameter update process is formulated as a gradient descent optimization problem as shown in Equation 4.1. In this equation, w_{ji} represents the weight of the connection between neuron i and j, E denotes the global error, α denotes the learning rate, and ψ' is the first derivative of the activation function. The Δw_{ji} is computed by propagating the error back in the network through all the connections in the downstream of j to the output.

$$\Delta w_{ji} = -\alpha \frac{\partial E}{\partial w_{ji}} = \sum_{k \in DownStream(j)} \left(\alpha \psi' w_{kj} \frac{\partial E}{\partial w_{kj}} \right) \tag{4.1}$$

In the proposed methodology to construct AxNNs, we utilize two unique properties: (i) the ability to apportion global errors to local computations by using backpropagation, and (ii) the ability to self-heal local errors in the network during training. A detailed description of the principles behind AxNNs and their design are provided in Section 4.3.

4.3 AxNN: Approach and Design Methodology

Approximate Neural Networks (AxNNs) are neural networks whose constituent computations have been subject to approximations, resulting in improved energy efficiency with acceptable output quality. This section outlines the key ideas behind AxNNs and the proposed design methodology.

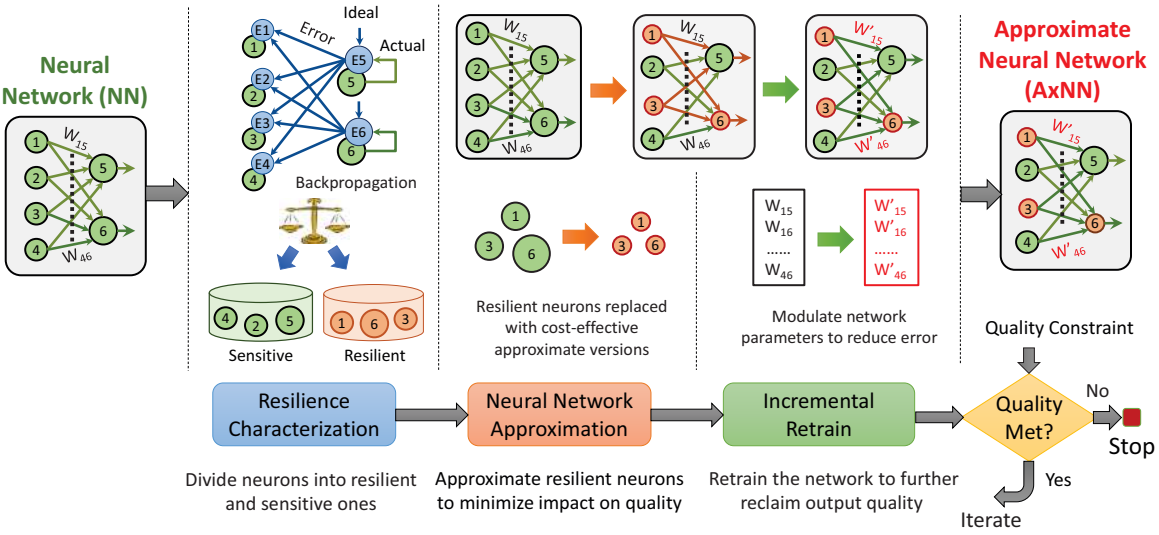


Fig. 4.5.: Overview of the Approximate Neural Networks (AxNN) design approach

4.3.1 AxNN: Design Approach

An AxNN can be viewed as a transformed version of a trained NN, where the transformation introduces approximations such that the resulting energy is minimized while the output quality meets a specified constraint. As shown in Figure 4.5, this transformation involves three key steps: (i) Resilience characterization, wherein the neural network is analyzed to identify neurons that impact output quality the least, (ii) Neural network approximation, in which the neurons that were determined to be resilient in the characterization step are approximated, and finally (iii) Incremental retraining wherein the network is retrained with the approximations in-place such that the loss in quality is further minimized. The following subsections provide an in-depth description of each step in the process.

Neural network resilience characterization

A significant challenge to employing approximate computing in any application is to distinguish computations that the application output is highly sensitive to (and hence cannot be approximated) from resilient ones that may be subject to approxima-

tions. In the context of neuromorphic systems, we propose to utilize backpropagation to characterize the resilience of each neuron. Backpropagation apportions the error at the output of the NN to the outputs of individual neurons. Thereby, it provides a measure of the error contributed by each neuron to the outputs of the network. We make the following key observation: neurons that contribute the least to the global error are more resilient *i.e.*, more amenable to approximations. Conversely, neurons contributing the highest error during backpropagation are deemed sensitive.

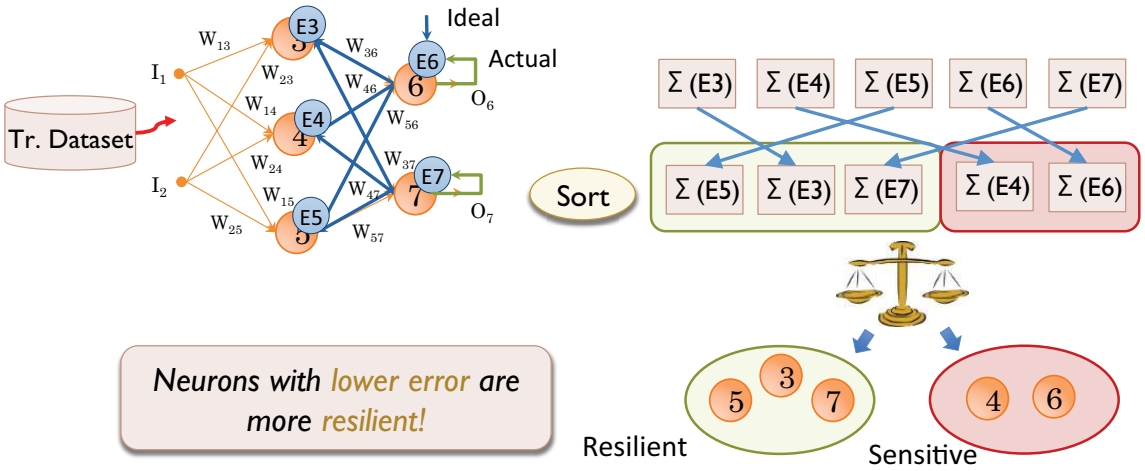


Fig. 4.6.: Illustration: Neuron resilience characterization

Based on the above insight, we propose a resilience characterization procedure, illustrated in Figure 4.6, that involves the following operations. For each instance in the training dataset, the error at the output of the neural network is computed using forward propagation. Next, the errors are propagated back to the outputs of individual neurons and their average error contribution over all inputs in the training set is obtained. The neurons are then sorted based on the magnitude of their average error contribution, and a pre-determined threshold is used to classify them as resilient or sensitive. We note that, unlike the actual training process, the network parameters are not altered during the resilience characterization step.

Approximation of resilient neurons

In the approximation step, the AxNN is formed by replacing *approximate neurons* in place of the resilient neurons identified during resilience characterization. Approximate neurons are inaccurate but cost-effective hardware or software implementations of the original neuron functionality and are the primary source of the energy efficiency in AxNNs. Approximate neurons can be designed using a wide range of approximate computing techniques. In this work, as shown in Figure 4.7, we explore two techniques to approximate resilient neurons. First, we utilize precision scaling, a popular technique in which the precisions (bit-widths) of the input operands and the neuron weights are modulated based on their degree of resilience. In addition, we also explore the use of piecewise-linear approximations of the activation function. These approximations may lead to improved efficiency on various hardware platforms. However, the proposed QCNPE architecture, described in Section 4.4, is specifically designed to translate the reduced precision requirements of the approximate neurons into energy improvements.

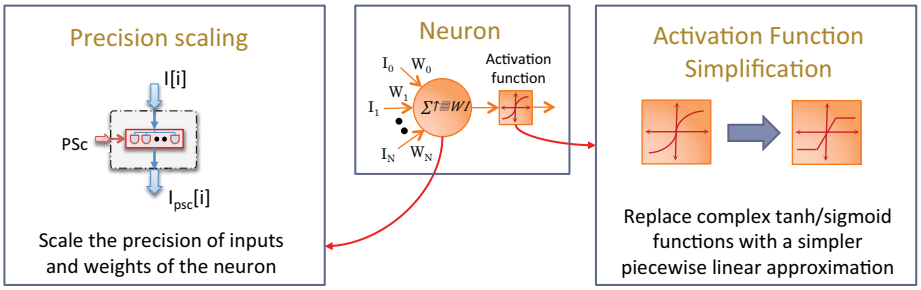


Fig. 4.7.: Techniques used to approximate neurons

Incremental retrain of AxNN

Although approximations are themselves introduced in a quality-aware manner, we show how to further minimize their impact by leveraging the training process. As discussed in Section 4.2, the training process modulates the parameters associated

with each neuron such that the global error is minimized. In fully-accurate NNs, the output error originates from untrained or partially trained network parameters. However, in the case of AxNNs, we intentionally supplement this error with a secondary source, *viz.* approximate neurons. Since training by nature has the ability to minimize errors at neuron outputs, we assert that errors introduced by approximations can also potentially benefit from it. Leveraging this insight, we propose to retrain the AxNN parameters with approximations in-place. The retraining process, as shown in Figure 4.8, suitably adjusts the AxNN parameters, thereby alleviating the impact of approximation-induced errors. Since retraining improves the output quality of the AxNN, it enables new opportunities to perform additional approximations. This synergy between approximation and training can be captured in an iterative approximate-and-retrain loop, as described in the next sub-section.

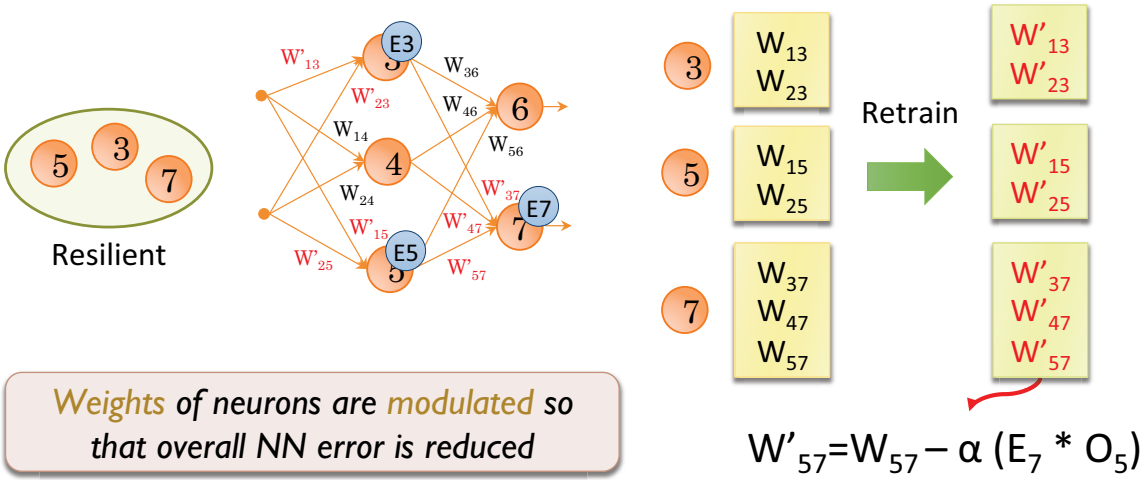


Fig. 4.8.: Incremental retrain of AxNN

Retraining the AxNN after approximations increases the overall runtime of the training process. However, we note that the retraining is incremental *i.e.*, it is carried out for very few iterations (2 iterations in our experiments). Typically, the training process in NNs takes several tens to hundreds of iterations and therefore the increase in run-time complexity due to retraining is small. Also, in typical use cases of NN applications, the training process is performed once or very infrequently. On the other

hand, the testing or evaluation phase, in which the actual classification is performed using the NN, extends for much longer periods of time. Since AxNNs yield significant energy benefits in the more critical evaluation phase, a small increase in the cost of training is a favorable trade-off. The impact of retraining on the overall energy and quality of AxNNs for different applications is discussed in Section 4.6.4.

4.3.2 AxNN Design Methodology

Algorithm 5 describes the pseudocode of the systematic methodology that we propose to automatically construct AxNNs. The inputs are a pre-trained neural network (NN), its corresponding training dataset ($TrData$) and a quality constraint (Q) that dictates the degradation in quality tolerable in the approximate implementation. The quality specifications are application-specific and are typically used during the process of constructing and training the NN itself. The algorithm iteratively builds the AxNN by successively approximating the NN in each iteration (lines 3-15), while ensuring that the quality bounds are satisfied.

The following steps are performed in each iteration of Algorithm 5. First, an estimate of energy consumed by each layer of the NN ($Layer.E_{List}$) is computed (line 5). For this purpose, we employ a high-level energy model of the quality-configurable neuromorphic processing engine discussed in Section 4.4. However, other energy models based on the complexity of neurons and the density of interconnections can also be utilized. We thus identify the most energy-intensive layer ($Layer_{Emax}$) in the network (line 6) and target its constituent neurons for approximations. Next, the resilience of each neuron in $Layer_{Emax}$ is characterized by finding the average error at its output over the entire training set using backpropagation (line 7). We then compute the mean of these errors (Δ_{mean}) and neurons whose error is below a threshold $\alpha * \Delta_{mean}$ are deemed resilient. Each of the resilient neurons previously identified are approximated in steps by gradually reducing their precision of computations (lines 9-13). The approximate neural network ($AxNN$) is thus obtained. Next, the $AxNN$ is in-

Algorithm 5 AxNN: Design methodology

Input: Pre-trained neural network: NN,

Training dataset: $TrData$, Quality constraint: Q

Output: Approximate neural network: $AxNN$

Begin
  Initialize: $AxNN_{temp} = NN$
  while $Q_{AxNN} > Q$ **do**
    $AxNN = AxNN_{temp}$
    $Layer.E_{List} \leftarrow$ Energy estimates of $AxNN$ layers
    $Layer_{Emax} \leftarrow \max\ (Layer.E_{List})$
    $Layer_{Emax}.\Delta =$ **backpropagation** ($AxNN$,$TrData$)
    $\Delta_{mean} =$ **mean** ($Layer_{Emax}.\Delta$)
    for each N: Neuron $\in Layer_{Emax}$ **do**
      if $Layer_{Emax}.\Delta(N)$ ¡ $\alpha * \Delta_{mean}$ **then**
        $AxNN_{temp} =$ Approximate N in $AxNN$
      end if
    end for
    $AxNN_{temp} =$ train ($AxNN_{temp}$, $TrData$, K epochs)
  end while
  return $AxNN$
End

crementally retrained for a small number of iterations (K epochs) to further improve its quality (line 14). After retraining, if the $AxNN$ meets the specified quality constraint, then lines 4-14 are repeated and the network is further approximated. If not, the last valid $AxNN$ is produced as the output.

The above design methodology can be utilized to construct energy-efficient approximate versions of any neural network, subject to the desired quality requirements. Furthermore, any approximation technique may be used, although approximations that result in better energy *vs.* quality tradeoffs are clearly desirable.

4.4 Quality Configurable Neuromorphic Processing Engine

In this section, we describe the proposed quality configurable Neuromorphic Processing Engine (QCNPE) that provides a hardware platform to execute AxNNs. The QCNPE is a many-core architecture that exploits the fine-grained data parallelism and data re-use patterns of NNs. A key feature of QCNPE is that it contains specialized processing elements whose accuracies (and energy) are dynamically configurable and hence can be used to efficiently execute neurons with various degrees of approximation.

Figure 4.9 shows the block diagram of QCNPE. It contains 2 types of processing elements: (i) a 2D array of neural compute units (NCUs), and (ii) a 1D array of activation function units (AFUs). The NCUs contain a 2-level datapath with an accumulator register and compute the weighted sum of a stream of inputs over multiple cycles. The NCUs are connected to their nearest neighbors in the 2D array and receive inputs from the left and top NCUs, which are then propagated to their right and bottom neighbors in the next cycle. The NCUs along the top and left borders of the 2D array receive inputs from two 1D arrays of First-In-First-Out (FIFO) memory elements placed along the borders. Functionally, the NCUs are designed to perform the weighted sum operation associated with each neuron. For this purpose, the inputs are streamed in along the rows and weights along the columns and operated

upon within each NCU. Note that the inputs and weights are re-used by all NCUs in a given row and column respectively, which is a typical data flow pattern in NNs, wherein the inputs fan-out to several neurons and the weights are shared amongst neurons/across inputs.

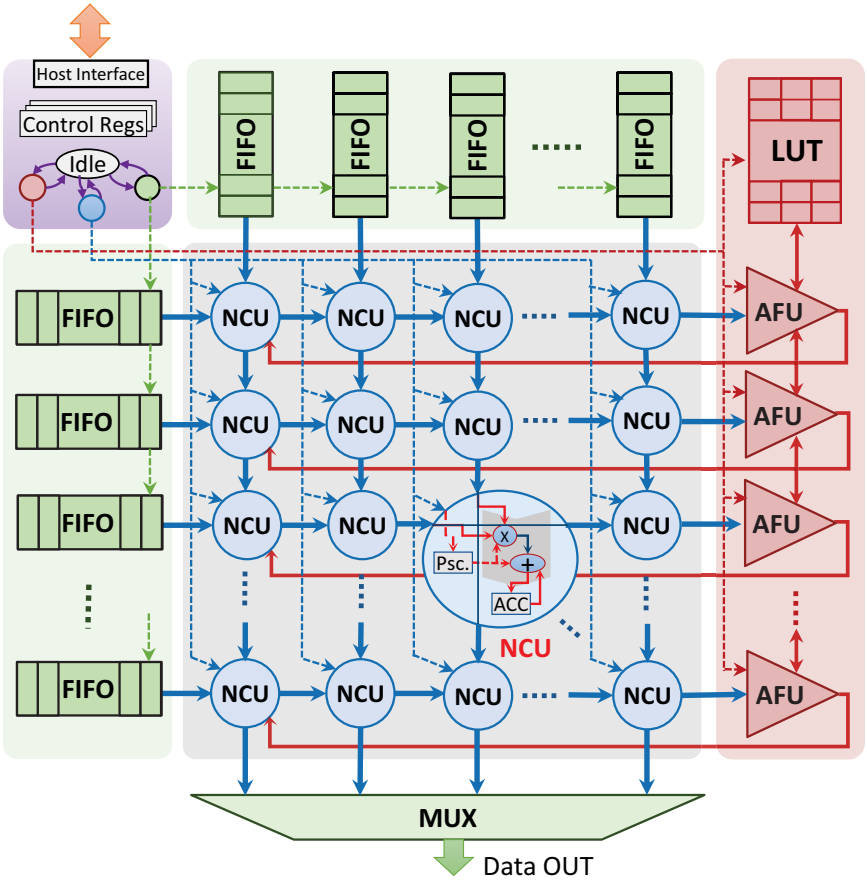


Fig. 4.9.: Block diagram of QCNPE

In order to facilitate execution with different accuracies, the NCUs are designed with a *precision control register*, which is initialized at the beginning of the 2D array operation. This is used to modulate the precision of the NCU inputs before they are operated within the NCU. Scaling the precision of input operands naturally results in power savings due to the reduction in switching activity in the NCU. In QCNPE, this is further enhanced by clock gating the LSB bit slices of the NCU accumulator

register. In our implementation, the area and power overheads to enable quality configurability amounted to less than 5% of the overall NCU.

The activation function units (AFUs), located on the right border of the 2D array, are designed to perform the non-linear operation on the weighted sum computed in the NCUs. As shown in Figure 4.9, this is carried out in a cyclical fashion, wherein the weighted sums from the NCUs in each row are streamed out and the outputs of the AFUs are stored back to the respective elements.

In summary, the QCNPE architecture provides an energy-efficient hardware platform to execute AxNNs of any given topology, interconnectivity pattern and degrees of approximation in their neurons.

4.5 Experimental Methodology

This section describes the experimental methodology and the benchmarks used in our evaluation of AxNNs. The QCNPE was implemented at the Register-Transfer Level (RTL) in Verilog HDL and mapped to the IBM 45nm technology using Synopsys Design Compiler. Synopsys Power Compiler was used to estimate the energy consumption of the implementation. The key micro-architectural parameters and implementation metrics are shown in Table 4.1.

Micro-architectural Parameters	Value
Array Dimension	16 X 16
No. of NCU/AFU	256/16
FIFO count	32
FIFO depth	32

Metric	Value
Feature Size	45nm
Area	1.7 mm^2
Power	517.2 mW
Gate Count	390392
Frequency	1GHz

Table 4.1.: QCNPE parameters and metrics

Neural networks used in 6 popular classification and recognition applications, listed in Table 4.2, were used as benchmarks in our experiments. The number of layers, neurons and connections in the networks are also provided in Table 4.2. The

benchmarks were ported manually to QCNPE and the baseline was well optimized for energy. We utilized classification accuracy, *i.e.,* the fraction of instances correctly classified as the measure of quality for all the benchmarks.

Applications	Dataset	Layers	Neurons	Connections
House Number Recognition	SVHN	8	47818	799616
Object Classification	CIFAR	6	38282	808608
Digit Recognition	MNIST	6	8010	43036
Face Detection	YUV faces	4	13362	25552
Object Recognition MLP	CIFAR	2	1034	3155968
Census Data Analysis	Adult	2	12	160

Table 4.2.: NN benchmarks used to evaluate AxNN

4.6 Results

In this section, we present the results of experiments that demonstrate the energy efficiency offered by AxNNs.

4.6.1 Energy benefits of AxNN

Figure 4.10 shows the energy improvement obtained using AxNNs for various output quality (classification accuracy) constraints. The energy of each AxNN is normalized to a fully-accurate QCNPE implementation in which none of the neurons are approximated. Note that this is already a highly optimized baseline since the QCNPE architecture is highly customized to the characteristics of NNs. Across all benchmarks, AxNN consistently provides significant energy benefits between 1.14X-1.92X for virtually no loss (¡ 0.5%) in application output quality. When the quality constraints are relaxed to ¡ 2.5% and ¡ 7.5%, the benefits increase to 1.35X-1.95X and 1.41X-2.3X, respectively. On an average, AxNN achieves 1.43X, 1.58X and 1.75X improvement in application energy for the different quality constraints.

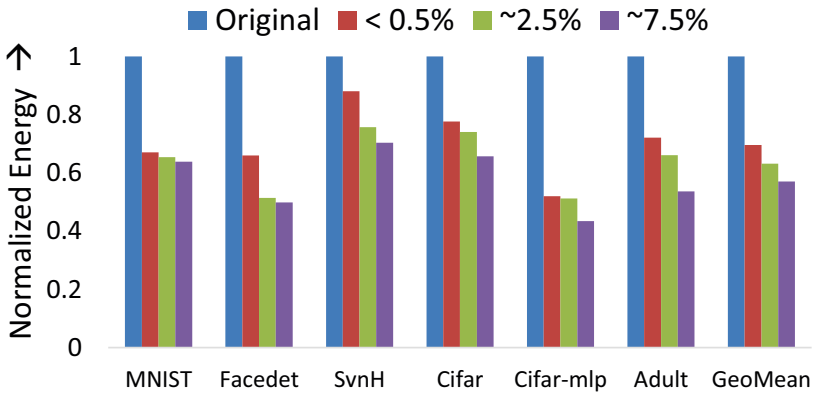


Fig. 4.10.: Improvement in energy using AxNN

4.6.2 Uniform approximation: Comparison

We now illustrate the effectiveness of the proposed resilience characterization methodology for NNs, by comparing it with a naïve approach wherein all the neurons in the NN are approximated uniformly. Figure 4.11 shows the energy *vs.* accuracy trade-off curves thus obtained for 3 different applications. We observe in all three cases that the energy improvement obtained using AxNNs is substantially better at all quality levels, compared to uniform approximation. Thus, it is critical to identify neurons that are amenable to approximations and directly applying approximate computing techniques without the proposed resilience characterization step would lead to limited benefits.

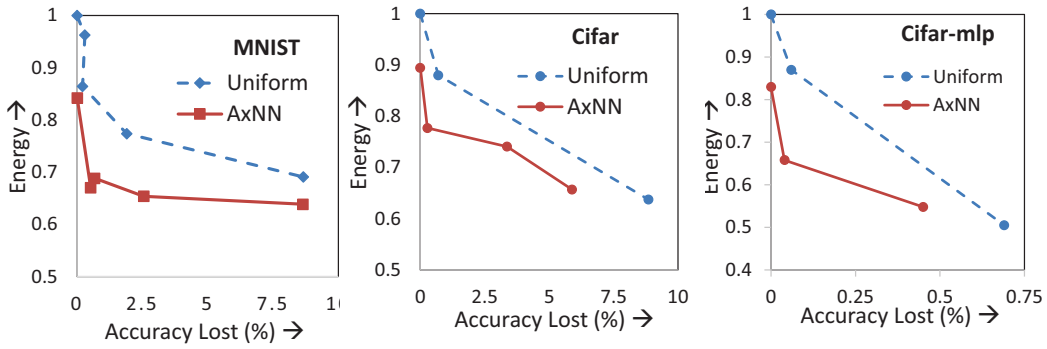


Fig. 4.11.: Quality *vs.* energy trade-offs with uniform and AxNN approximations

4.6.3 Resilience Characterization: Insights

We present insights into the process of identifying resilient neurons in NNs and illustrate them using the digit recognition application (MNIST) [39] as an example. The NN takes a pixel map of a handwritten digit as its input and classifies it amongst digits 0,1 . . . 9. The network contains 6 layers and progressively extracts feature maps from the input image in the first four layers and combines them in layers 5 and 6 to infer the class of the input. Each pixel in each feature map of each layer corresponds to the output of a neuron.

Figure 4.12 shows the average errors (obtained using backpropagation) at the outputs of all neurons in four selected layers of the digit recognition network. The neurons are color-coded (blue to red) based on the magnitude of their errors and are located on the feature maps corresponding to the pixel they generate. We observe that the resilience of the neurons varies widely (6 orders of magnitude) across all layers and to a substantial extent (4 orders of magnitude) within a given layer. We also find that the fraction of neurons that are resilient decreases sharply as we move closer to the NN outputs. This is attributed to the fact that neurons in the initial layers typically process features local to a certain region of the image, while neurons in the final layers infer global features from the previously extracted local features. Since errors in global inferences are less tolerable, the neurons in the final layers are correspondingly more sensitive. Further, errors in neurons closer to the inputs have a greater chance of being compensated or filtered-out as they propagate through the NN.

We also observe a significant correlation between the resilience of neurons and the region of the image on which they operate. For example, in layer 1 of Figure 4.12, neurons that process the center of the input image, where information is typically concentrated, are less resilient. The neurons become progressively more resilient when proceeding towards the borders of the image. Thus, the resilience characterization

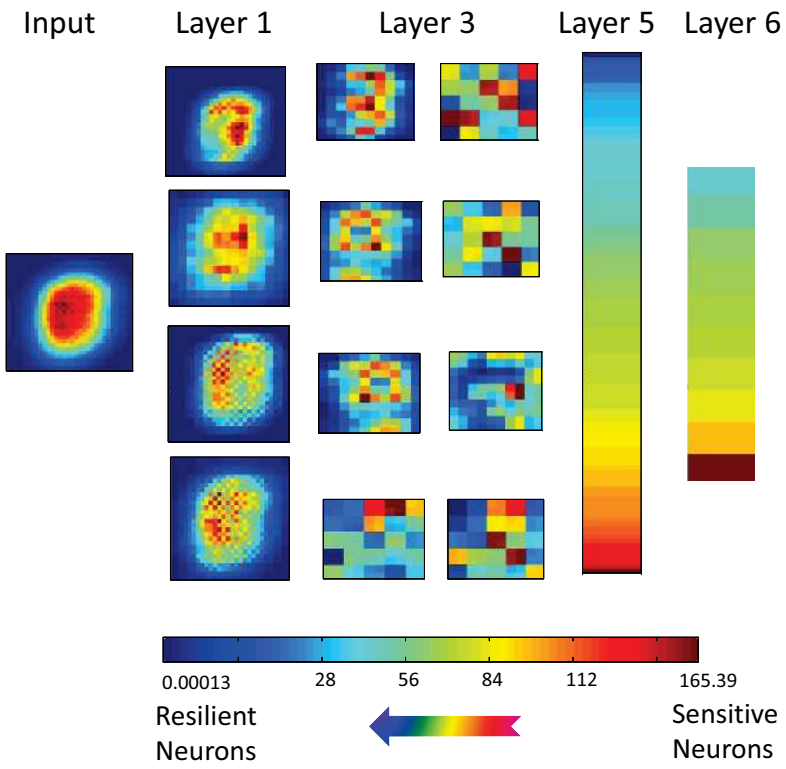


Fig. 4.12.: Neuron average error maps in MNIST [39]

methodology utilized in AxNNs captures the physical intuitions behind the resilience of neurons in NNs.

4.6.4 Impact of Retraining

To understand the benefits of incrementally retraining the network with the approximations in place, Figure 4.13 plots the normalized energy-quality trade-off obtained with and without the retraining step in the AxNN methodology for four applications. We observe in all four cases that AxNN with retraining provides a superior trade-off, *i.e.*, lower energy for a given target quality. This is because, retraining recovers a good amount of quality lost due to approximations, thereby allowing additional approximations for the same quality.

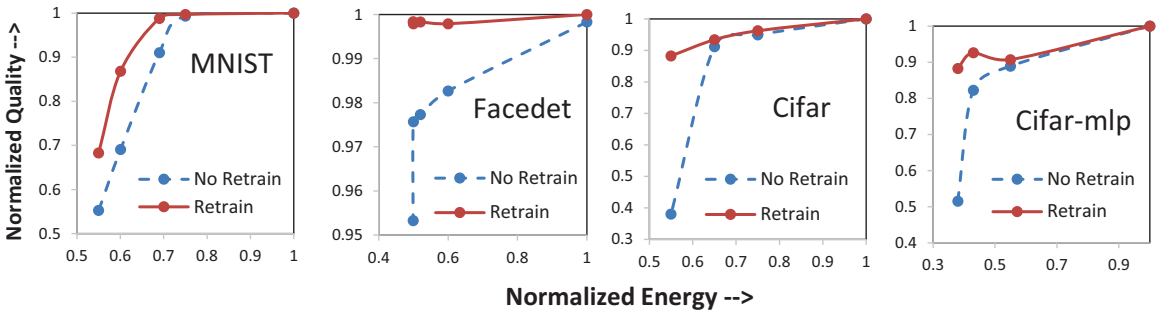


Fig. 4.13.: Impact of retraining on energy and quality

Across all our benchmarks, retraining the AxNN increased the run time of the training process on an average by 21.5%. As discussed in Section 4.3.1, we believe that this moderate increase in the training time is quite insignificant relative to the energy benefits provided during the energy-critical testing phase of the application.

4.6.5 AxNNs on Commodity Platforms

In the previous subsections, the neurons were approximated by scaling the precision of their input operands and the energy benefits were evaluated using the proposed QCNPE architecture. We now evaluate the benefits of AxNNs on commodity platforms by designing approximate software implementations of NNs. Towards this end, we replace the activation functions of selected neurons in the network, identified by the AxNN methodology, with an approximate but significantly faster piecewise linear function. The original and approximate implementations were executed on a server with an Intel Xeon processor at 2.7 GHz and 132 GB memory. We note that the software baseline implementation was aggressively optimized for performance.

Figure 4.14 shows the normalized runtime and quality of the software AxNN implementations, with varying fraction of neurons approximated, for three applications. The graphs reveal that, as the fraction of neurons approximated by the AxNN methodology increases, the runtime decreases proportionally. However, the corresponding decrease in the application output quality is disproportionately small

due to the careful selection of neurons and re-training. On an average, the runtime speedup is 1.35X with $< 0.5\%$ loss in the output quality. These results underscore the generality of the AxNN methodology with respect to both the approximate computing technique employed to create approximate neurons, as well as the hardware platform used for their execution.

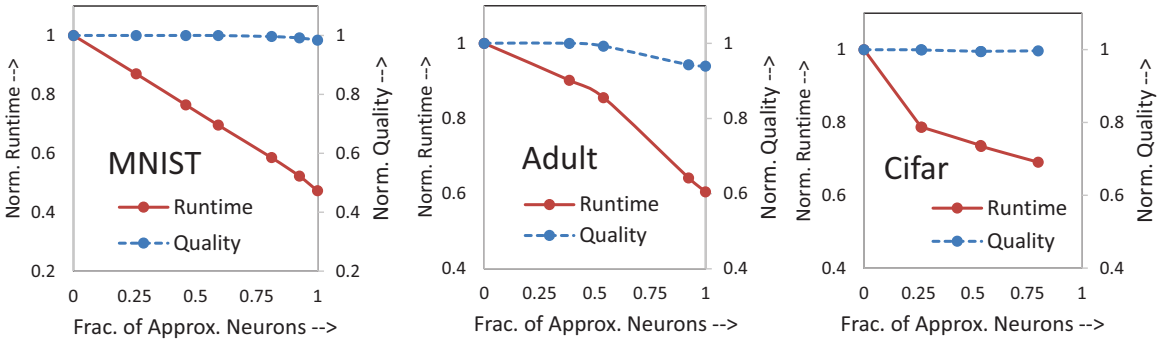


Fig. 4.14.: AxNN runtime on commodity platform

4.7 Summary

Neuromorphic systems are growing increasingly prevalent and are popularly employed in a wide variety of classification, recognition, search and computer vision applications. In this thesis, we utilize approximate computing, an emerging design paradigm, to design energy-efficient neuromorphic systems. We propose the concept of Approximate Neural Networks (AxNNs), in which neurons that impact output quality the least are systematically identified and approximated. The AxNN is then retrained with the approximations in place, leading to additional opportunities to further approximate the network. Also, we design a quality configurable neuromorphic processing engine that can be utilized to efficiently execute AxNNs. Our experiments on six NN applications demonstrated significant improvements in energy for negligible loss in the output quality.

5. SCALABLE EFFORT CLASSIFIERS

5.1 Introduction

For many computational systems, all inputs are not created equal. Consider the simple example of 8-bit multiplication; intuitively, computing the product of 02h and 01h should be easier than multiplying 19h and 72h. Similarly, compressing a picture that contains just the blue sky should take less effort than one that contains a busy street. Ideally, to improve both speed and energy efficiency, algorithms should expend effort (computational time and energy) that is commensurate to the difficulty of the inputs. Unfortunately, for most applications, discriminating easy inputs from hard ones at runtime is challenging. Thus, hardware or software implementations tend to expend constant computational effort as determined by worst-case inputs or a representative set of inputs. In this chapter, we focus on an important class of algorithms - machine learning classifiers - and show how they can be constructed to scale their computational effort depending on the difficulty of the input data, leading to faster and more energy-efficient implementations.

Machine-learning algorithms are used to solve an ever-increasing range of classification problems in recognition, vision, search, analytics and inference across the entire spectrum of computing platforms [66]. Machine learning algorithms operate in two phases: training and testing. In training, decision models are constructed based on a labeled training data set. In testing, the learnt model is applied to classify new input instances. The intuition behind our approach is as follows. During the training phase, instead of building one complex decision model, we construct a cascade or series of models with progressively increasing complexity. During testing, depending on the difficulty of an input instance, the number of decision models applied to it is varied, thereby achieving scalability in time and energy.

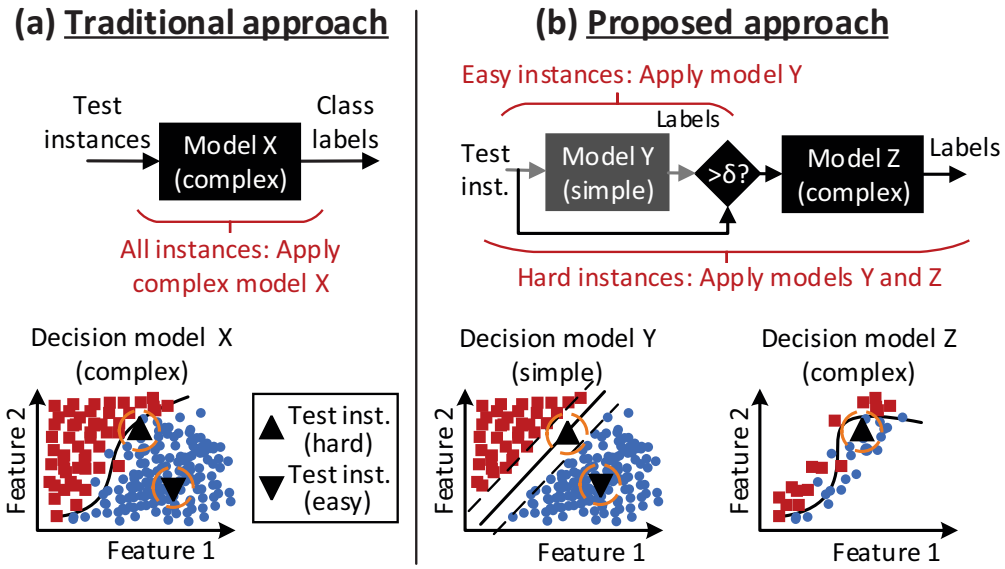


Fig. 5.1.: Scalable effort classifiers: Approach

Figure 5.1 illustrates our methodology through a specific machine learning algorithm namely a binary support-vector machine (SVM) classifier. In the traditional approach shown in Figure 5.1(a), input training examples are used to build a decision boundary (denoted as model X) that separates data into two categories or classes. At test time, data instances are assigned to one class or the other depending on their location relative to the decision boundary. The computational effort (in terms of energy and time) to process every test instance depends on the complexity of the decision boundary, *e.g.*, non-linear boundaries typically require large number of support vectors, and hence cost more than linear ones. In the example of Figure 5.1(a), a single model (X) clearly needs to use a non-linear boundary in order to separate the classes with high accuracy. However, this leads to high computational effort for not only the hard test data instances (points close to the decision boundary) but also the easy test data instances (points far away from the decision boundary). In contrast, Figure 5.1(b) shows our approach, where we create multiple decision models (Y and Z) with varying levels of complexity. In the simpler model (Y), two different linear decision boundaries (dashed lines) are used to define a region around the original

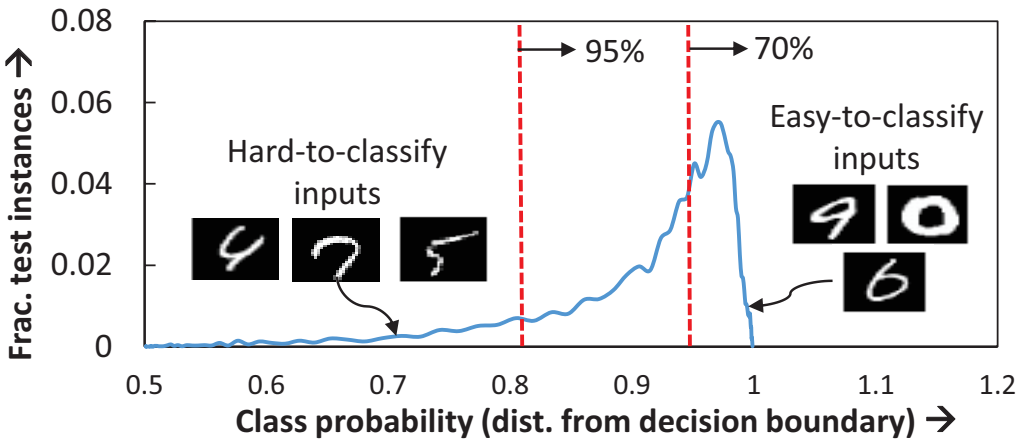


Fig. 5.2.: Distribution of class probabilities for MNIST dataset

non-linear boundary, and selectively classify the easier training instances that lie beyond this region. The complex model (Z) is employed only for instances that cannot be classified by the simpler model. This approach can save time and energy, since all data instances need not be processed by the more complex non-linear decision model.

The amount of computational time and energy saved depends on the distribution of input data. Fortunately, in many useful applications, lots of test data is easy. For instance, while detecting movement using a security camera, most video frames contain only static objects. We quantify this intuition for the popular MNIST handwriting recognition dataset [67]. Figure 5.2 shows the distribution of the probability with which the classifier predicted the test instances to a particular class in the dataset (inset images show some representative hard and easy instances). Observe that over 70% of the inputs were predicted with > 0.95 certainty. The fraction increases up to 95% at a probability > 0.8. Therefore, a significant majority of inputs lie far away from the decision boundary, and a low complexity classifier can be employed in their context.

We generalize the approach described above for any machine learning classifier by constructing a cascaded series of classification stages with progressively increasing

complexity and accuracy. We also show how to construct simpler models for each stage by using an ensemble of biased classifiers.

How do we determine the difficulty of an instance at runtime? Besides model partitioning, this is another challenge that we address in the chapter. We determine the hardness of each test data instance implicitly. The top portion of Figure 5.1(b) illustrates our approach. We process test instances through the decision models in a sequence starting from the simplest model. After the application of every model, we estimate the confidence level of the produced output (*i.e.*, the class probability or classification margin). Constructing each model as an ensemble of biased classifiers further facilitates this, since their consensus may be used to indicate the confidence of classification. If the confidence is above a threshold, we accept the output class label produced by the current model and terminate the classification process. Simpler data instances get processed by only the initial few (simpler) models, while harder instances need to go through more models. Thus, our approach provides an inbuilt method to scale computational effort dynamically.

In summary, we make the following contributions:

- Given any machine learning classifier, we propose a systematic approach to construct a scalable effort version thereof by cascading classification stages of growing accuracy and complexity. The scalable effort classifier has accuracy comparable to the original one, while being faster and more energy efficient.

- To construct the stages of the scalable effort classifier, we propose ensembles of biased classifiers and a consensus operation that determines the confidence level in the class labels produced by the classification stage.

- We present an algorithm to train a scalable effort classifier that trades off the number of stages, complexity of each stage, and fraction of inputs classified by each stage, to optimize the overall computational effort spent in classification.

- Across a benchmark suite of eight applications that utilize 3 classification algorithms, we show that scalable effort classifiers provide 1.5× average reduction

in energy. Through hardware implementations in a 45nm SOI process, we also demonstrate an average of 2.3× reduction in energy.

The rest of the chapter is organized as follows. In Section 5.2, we describe our approach to the construction of scalable effort classifiers. In Section 5.3, we describe a methodology to construct such classifiers. In Section 5.4, we describe our evaluation methodology and benchmarks. We present experimental results in Section 5.5 and conclude in Section 5.6.

5.2 Scalable effort Classifiers

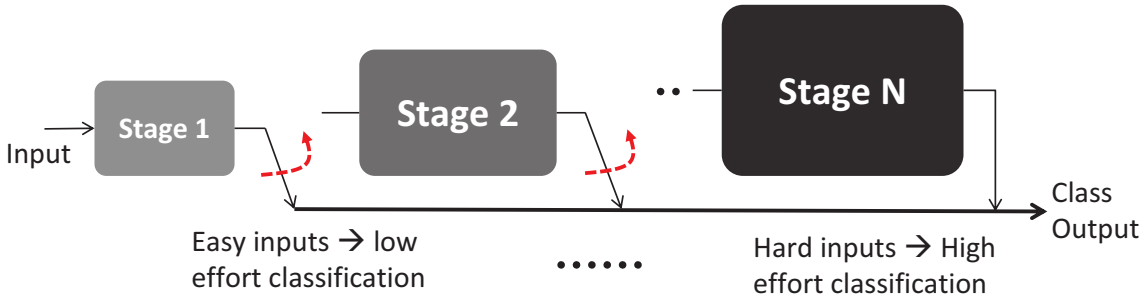


Fig. 5.3.: A scalable effort classifier consists of a sequence of decision models, which grow progressively more complex

In this section, we present our structured approach to design scalable effort classifiers. Figure 5.3 shows the conceptual view of a scalable effort classifier. Given any classification algorithm, different models are learnt using the same algorithm and training data. These models are then connected in a sequence such that the initial stages are computationally efficient but have lower classification accuracies, while the later ones have both higher complexities and accuracies. Further, each stage in the cascade is also designed to implicitly assess the hardness of the input. During test time, data is processed through each stage, starting from the simplest model, to produce a class label. The stage also produces a confidence value associated with the class label. This value determines whether the input is passed on to the next stage or not. Thus, class

labels are produced earlier in the chain for easy input instances and later for the hard ones. If an instance reaches the final stage, the output label is used irrespective of the confidence value. Next, we present more details on how each stage of Figure 5.3 is designed.

5.2.1 Design of Scalable effort Classifier Stage

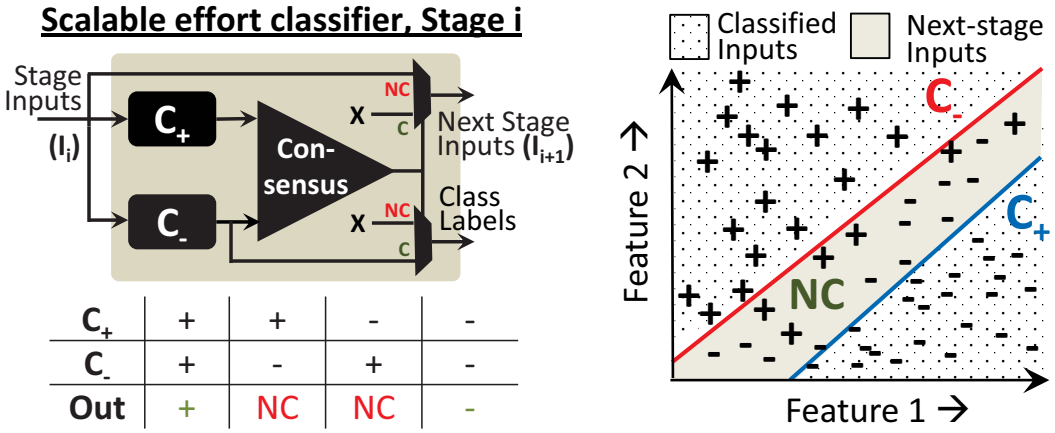


Fig. 5.4.: Design of scalable effort classifier stage

First, we consider the case of a binary classifier with two possible class outcomes + and -. Figure 5.4a shows the block diagram of a classifier stage. In such a scenario, each stage is composed of two biased classifiers, which are trained to detect one particular class with high accuracy. For instance, if a classifier is biased towards class + (denoted by C_+), it frequently mispredicts inputs from class - but seldom from class +. Besides the biased classifiers, the stage also contains a consensus module, whose functionality is shown in Figure 5.4a. The consensus module utilizes the output of the biased classifiers to determine if the input should get classified in the current stage or passed on to the next stage. This decision is based on the following two criteria:

1. If the biased classifiers predict the same class *i.e.*, ++ or - -, then the corresponding label *i.e.*, + or - is produced as output.

2. If the biased classifiers produce no consensus (NC) *i.e.*, +- or -+, the input is deemed to be difficult to classify by the stage and is passed along to the next stage.

To better understand how each stage functions, consider the example shown for a binary SVM in Figure 5.4b. The two biased classifiers used (*i.e.*, C_+ and C_-) are linear SVMs, which are computationally efficient. Observe how the decision boundaries for the two classifiers are located such that they do not misclassify instances from the class towards which they are biased. For all input test instances that lie in the hatched region, both biased classifiers provide identical class labels (*i.e.*, consensus). However, there is no consensus on input instances that lie in the grayed-out region. Test instances in this latter region are thus passed on to the next stage. In summary, the biased classifiers define a region around the original decision boundary that separates the easy *vs.* hard inputs. The consensus operation determines which region an input lies in, and selectively classifies the input or passes it to the next stage.

5.2.2 Efficiency and Accuracy Optimization

As scalable effort classifiers are composed of many individual stages in a sequence, the following two factors determine their overall efficiency and accuracy: (1) the number of connected stages and (2) the fraction of inputs that is processed by each stage. These factors present a fundamental tradeoff in the design of scalable effort classifiers, which is discussed in this subsection.

Adding Stages to a Scalable effort Classifier

First, we analyze when it is desirable to add a new stage to the scalable effort cascade. Consider a classifier stage i, with computational cost γ_i per instance. Let I_i be

the fraction of inputs that reach stage i. The stage classifies a subset of these inputs, while passing a smaller fraction (I_{i+1}) to the next stage. The condition described in Equation 5.1 should be satisfied for stage i to improve the overall efficiency of the scalable effort cascade.

$$(\gamma_{i+1} - \gamma_i) \cdot (\mathrm{I}_i - \mathrm{I}_{i+1}) > \gamma_i \cdot \mathrm{I}_{i+1} \tag{5.1}$$

The left-hand side of Equation 5.1 represents the improvement in efficiency due to the stage, which is the product of the fraction of inputs it classifies (I_i-I_{i+1}) and the reduction in complexity compared to the next stage. This should be greater than the right-hand side of Equation 5.1, which quantifies the penalty the stage imposes on inputs that it fails to classify *i.e.* if the stage were not present then those inputs (I_{i+1}) could be directly classified by stage $i+1$.

Consensus Threshold

The consensus operation described in Section 5.2.1 also influences the efficiency and accuracy of the scalable effort classifier. Most classification algorithms, in addition to the class output, provide a measure of confidence (*e.g.* class probabilities, distance from decision boundary *etc.*) in the prediction. We combine the class outputs of the biased classifiers along with their confidence measures to define a new consensus operation (Equation 5.2) that allows us to more directly control the efficiency and accuracy of the cascade.

$$ConsOut = \begin{cases} \mathrm{C}_+ & \text{if } \mathrm{C}_+ == \mathrm{C}_-,\ \delta \geq 0,\ |\mathrm{C}_+| \text{ and } |\mathrm{C}_-— > \delta \\ \mathrm{C}_+ & \text{if } \mathrm{C}_+ == \mathrm{C}_-,\ \delta < 0 \\ \mathrm{C}_+ & \text{if } \mathrm{C}_+ \neq \mathrm{C}_-,\ \delta < 0,\ |\mathrm{C}_+| \text{ and } |\mathrm{C}_-| > \delta,\ |\mathrm{C}_+| \geq |\mathrm{C}_-| \\ \mathrm{C}_- & \text{if } \mathrm{C}_+ \neq \mathrm{C}_-,\ \delta < 0,\ |\mathrm{C}_+| \text{ and } |\mathrm{C}_-| > \delta,\ |\mathrm{C}_-| > |\mathrm{C}_+| \\ NC & \text{otherwise} \end{cases} \tag{5.2}$$

As shown in Equation 5.2, the consensus operation contains a parameter called the *consensus threshold* (denoted by δ) that the defines the degree to which the biased

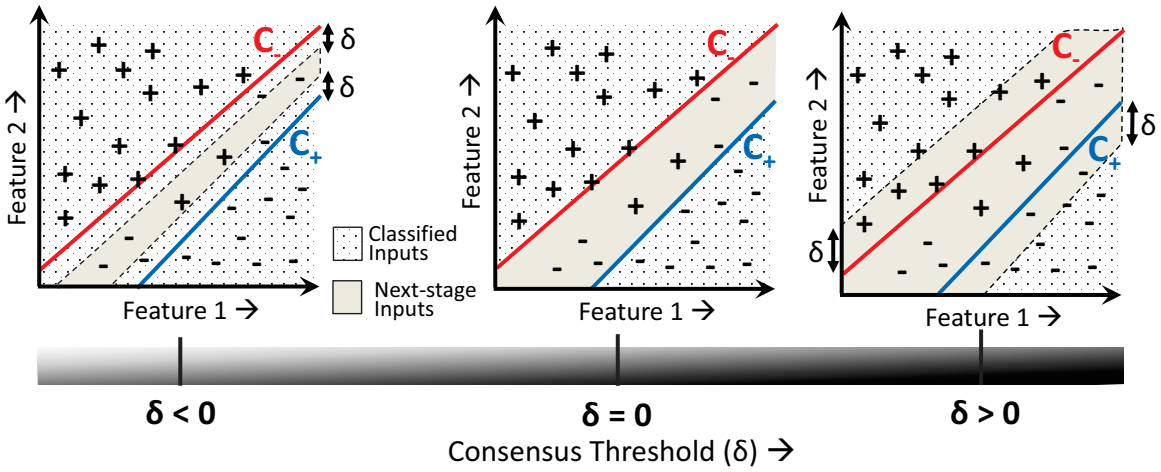


Fig. 5.5.: δ controls the fraction of inputs classified by a stage.

classifiers should agree (or contradict) for the input to be classified (or passed on) by the stage. A positive δ makes the consensus operation more stringent, *i.e.,* an input is classified by the stage only if the biased classifiers agree on their decisions and their respective confidence measures are greater than δ. For a positive δ, fewer inputs will be classified by the stage, but the accuracy of its classifications is improved. On the other hand, a negative δ relaxes the consensus threshold, as an input is classified by the stage even if the biased classifiers disagree, provided their confidence in the contradictory predictions is lower than δ. Figure 5.5 illustrates the impact of different choices of δ for the binary SVM classifier considered in Section 5.2.1. In this case, we observe that modulating δ grows or shrinks the region separating the easy *vs.* hard inputs, resulting in the stage classifying a correspondingly smaller or larger fraction of inputs. For computational efficiency, we choose the smallest value of δ at training time that yields no misclassifications.

Building the Component Classifiers

The final factor that determines the efficiency and accuracy of the scalable effort cascade is how the biased classifiers in each stage are built. We employ the following methods to build the component classifiers.

Modulating complexity through algorithmic knobs: Many classification algorithms inherently contain parameters that modulate their complexity and accuracy. Some examples include changing the kernel function of SVM, and the number of neurons and layers in a neural-network. We modulate these parameters to progressively increase complexity as new stages are added.

Biasing classifiers by asymmetric weighting, resampling and sub-sampling: To bias classifiers, we assign larger misclassification penalties to training instances of a given class. Alternatively, biasing can be implicitly achieved by generating additional instances for a class by adding some noise to the existing instances, or by sub-sampling instances from the opposite class. Note that biasing also influences the complexity of the classifier.

5.2.3 Multi-way Scalable effort Classifiers

We extend our approach to multi-class problems by employing a well-known strategy called one *vs.* rest classification, which reduces the computation to multiple binary classifications. The strategy involves training one classifier per class, with samples from that class regarded as positive (*i.e.*, +) while the rest are negative (*i.e.*, -). At test time, the highest confidence value across multiple such one *vs.* rest classifiers determines the final class assignment.

Figure 5.6(a) shows the design of a stage of a scalable effort multi-way classifier. It comprises several binary classification units, each containing a pair of biased classifiers and a local consensus (LC) module similar to the one shown in Figure 5.4a. It also contains a global consensus (GC) module, which aggregates outputs from all LC

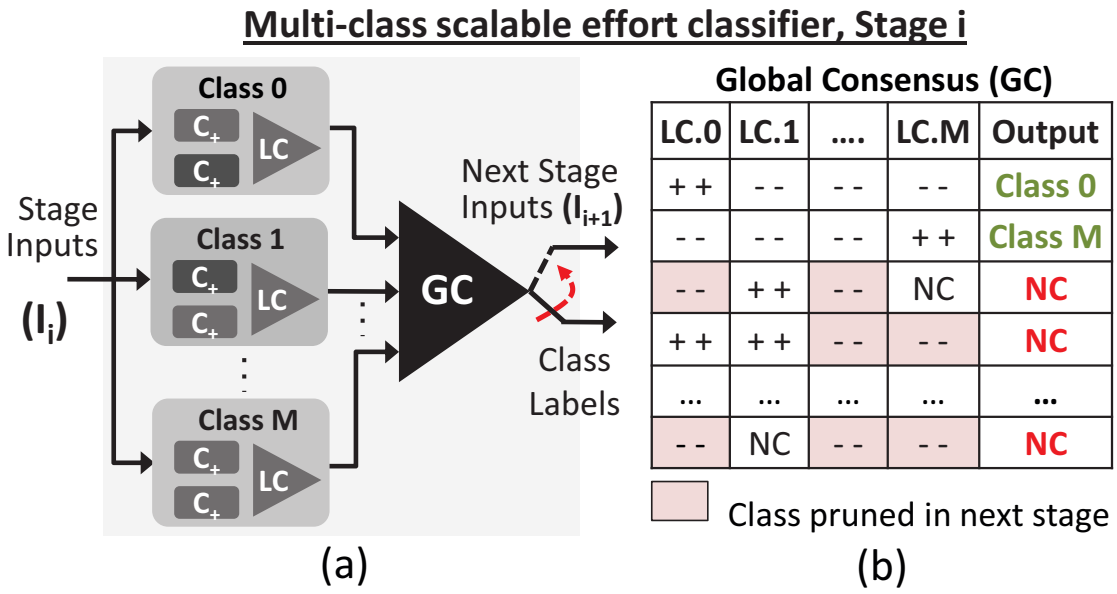


Fig. 5.6.: One *vs.* rest approach is used for multi-way classification. GC can prune some classes in the next stage.

modules in the stage. The functionality of GC is illustrated in Figure 5.6(b). If there is positive consensus (*i.e.*, ++) in exactly one LC module, then the GC outputs a class label corresponding to the consenting binary-classification unit. If more than one LC module provides consensus, then the next stage is invoked.

Another feature of multi-way scalable effort classifiers is *class pruning*, *i.e.*, even if a stage does not classify a given input, it can eliminate some of the classes from consideration in the next stage. Specifically, if there is no consensus in the GC module and if the LC output shows negative consensus (*i.e.*, - -) then binary classification units corresponding to that particular class need to be evaluated in subsequent stages. Thus, only classes that produce positive consensus or NC are retained down the chain. This early class pruning leads to increased computational efficiency.

5.3 Design Methodology

In this section, we describe the procedure for training and testing scalable effort classifiers.

5.3.1 Training Scalable effort Classifiers

Algorithm 6 shows the pseudocode for training. The process takes the original classification algorithm C_{orig}, training data D_{tr}, and number of classes M as input. It produces a scalable effort version of the classifier C_{se} as output, which includes the biased classifiers $C_{+/-}$ and consensus thresholds δ for each stage.

First, we train C_{orig} on D_{tr} and obtain its cost γ_{orig} (line 1). Then, we iteratively train each stage of the scalable effort classifier C_{stg} (lines 2-22). The algorithm terminates if a stage does not improve the overall gain G_{stg} beyond a certain threshold ϵ (line 3). Next, we describe the steps involved in designing each stage of C_{se}.

To compute C_{stg}, we initialize G_{stg} to $+\infty$, and a classifier complexity parameter λ_{stg} to its minimum value (line 2). Then, we obtain $C_{+/-}$ (line 5). We follow-up by assigning the smallest value of δ that yields no misclassifications on D_{tr} to be the consensus threshold for the stage δ_{stg} (line 6). Once we determine $C_{+/-}$ and δ_{stg} for all classes, we proceed to estimate the number of inputs classified by the stage ΔI_{stg} by iterating over D_{tr} (line 9-17). During this time, we compute LC and GC values for each instance in D_{tr} (lines 10-11). For any instance, if global consensus is achieved (line 12), we remove it from D_{tr} for subsequent stages and increment ΔI_{stg} by one (line 13). If not, we add a fractional value to ΔI_{stg}, which is proportional to the number of classes eliminated from consideration by the stage (line 15). After all instances in D_{tr} are exhausted, we compute G_{stg} as the difference between the improvement in efficiency for the inputs it classifies and the penalty it imposes on inputs that it passes on to the next stage (line 18). We admit the stage C_{stg} to the scalable effort classifier chain C_{se} only if G_{stg} exceeds ϵ (line 19). Since instances that are classified by the stage are removed from D_{tr} used for subsequent stages, one or more classes may be

Algorithm 6 Methodology to train scalable effort classifiers

Input: Original classifier C_{orig}, training dataset D_{tr}, # classes M

Output: Scalable effort classifier C_{se} (incl. δ and $C_{+/-}$ $\forall$ stages)

1: Train C_{orig} using D_{tr} and obtain classifier cost γ_{orig}
2: **initialize** stage gain $G_{stg} = +\infty$, complexity param. $\lambda_{stg} = \lambda_{min}$, and allClassesPresent = true
3: **while** ($G_{stg} > \epsilon$ **and** allClassesPresent) **do**
4: **for** currentClass :=1 **to** M **do** *// evaluate stage C_{stg}*
5: Train $C_{+/-}$ biased towards currentClass using D_{tr} and λ_{stg}
6: $\delta_{stg} \leftarrow$ minimum δ s.t. no misclassifications in D_{tr}
7: **end for**
8: **initialize** # input instances to stage I_{stg} = # instances in D_{tr} and # instances classified by stage $\Delta I_{stg} = 0$
9: **for each** trainInstance $\in D_{tr}$ **do** *// compute ΔI_{stg} for C_{stg}*
10: Compute local consensus LC $\forall$ M classes
11: Compute global consensus GC
12: **if** GC $\leftarrow$ true **then**
13: remove trainInstance from $\in D_{tr}$ and $\Delta I_{stg} \leftarrow \Delta I_{stg} + 1$
14: **else**
15: $\Delta I_{stg} \leftarrow \Delta I_{stg}$ + # negative LCs / M
16: **end if**
17: **end for**
18: $G_{stg} = (\gamma_{orig} - \gamma_{stg}) \cdot \Delta I_{stg} - \gamma_{stg} \cdot (I_{stg} - \Delta I_{stg})$
19: **if** $G_{stg} > \epsilon$ **then** admit stage C_{stg} into C_{se}
20: **if** any class is absent in D_{tr} **then** allClassesPresent $\leftarrow$ false
21: λ_{stg} ++ *// increase classifier complexity for next stage*
22: **end while**
23: **append** C_{orig} as the final stage of C_{se}

exhausted. In this case, we terminate the construction of additional stages (line 20) and proceed to append the final stage (line 23). The complexity of the classifier is increased for subsequent stages (line 21).

5.3.2 Testing Scalable effort Classifiers

Algorithm 7 shows the pseudocode for testing. Given a test instance i_{test}, the process obtains the class label L_{test} for it using C_{se}. First, the list of possible outcomes is initialized to the set of all class labels (line 1). Each stage C_{stg} is invoked iteratively (lines 2-15) until the instance is classified (lines 2). In the worst case, C_{orig} is employed in the final stage to produce a class label (lines 3-4). In all other cases, the following steps are carried out. At each active stage, $C_{+/-}$ are invoked to obtain an estimate of LC (line 6) and GC (line 7). If global consensus is achieved, *i.e.*, one LC output is positive and the rest are negative (lines 8-10), then the instance is predicted to belong to the class with the positive LC value (line 9). If not, the list of active classes is pruned by removing the classes for which LC is negative (line 11). Subsequent stages are then invoked with the reduced set of possible outcomes (line 14).

In summary, C_{se} implicitly distinguishes between inputs that are easy and hard to classify. Thus, it improves the overall efficiency of any given data-driven classification algorithm.

5.4 Experimental Methodology

In this section, we describe our experimental setup used to evaluate the performance of scalable-effort classifiers.

5.4.1 Application Benchmarks

Table 5.1 shows the benchmarks and datasets that we use in our experiments. We evaluated 8 applications with up to over 9000 features and up to 10 classes. Between

Algorithm 7 Methodology to test scalable effort classifiers

Input: Test instance i_{test}, scalable effort classifier C_{se}, # stages N_{se} in C_{se}, and # possible classes M

Output: Class label L_{test}

1: **initialize** possibleClassesList = {1,2,...,M}, currentStage = 1, and instanceClassified = false
2: **while** instanceClassified = false **do**
3: **if** currentStage = N_{se} **then** *// apply C_{se} to i_{test}*
4: $L_{test} \leftarrow C_{se}$ [i_{test}]; instanceClassified $\leftarrow$ true
5: **else**
6: Compute local consensus LC $\forall$ possibleClassesList
7: Compute global consensus GC
8: **if** *GC* $\leftarrow$ true **then** *// global consensus achieved*
9: $L_{test} \leftarrow$ label $\in$ max (LC); instanceClassified $\leftarrow$ true
10: **else**
11: $\forall$ LC = -1, delete labels from possibleClassesList
12: **end if**
13: **end if**
14: currentStage $\leftarrow$ currentStage + 1
15: **end while**

them, they utilize three common supervised machine-learning algorithms, namely SVM, neural networks, and decision trees (J48 algo. [81]).

Table 5.1.: Application benchmarks used to evaluate scalable effort classifiers

Algorithm	**Application**	**Dataset [82]**	**Features / Classes**
Support Vector machines	Handwriting reco.	MNIST [67]	784 / 10
	Human activity reco.	Smartphones	561 / 6
	Eye detection	YUV faces	512 / 2
	Text classification	Reuters	9947 / 2
Neural networks	Enzyme classification	Protein	356 / 3
	Census data analysis	Adult	114 / 2
Decision trees-J48	Game prediction	Connect-4	42 / 3
	Census data analysis	Adult	114 / 2

5.4.2 Energy Evaluation

We implemented scalable-effort versions of each of the applications in C#. We also integrated WEKA, a machine-learning toolkit, as a backend to our software [83]. This helped us rapidly train and evaluate different classifiers. We measured runtime for the applications using performance counters on a commodity Intel Core i5 notebook with a 2.5 GHz processor and 8 GB of RAM. For the hardware implementation, we specified each classifier as an accelerator at the register-transfer logic (RTL) level. We used Synopsys design compiler to synthesize the integrated design to a 45 nm SOI process from IBM. Finally, we used Synopsys Power Compiler to estimate energy consumption of the synthesized netlists.

5.5 Results

In this section, we present experimental results that demonstrate the benefits of our approach.

5.5.1 Energy Improvement

Figure 5.7 shows the normalized improvement in efficiency with scalable effort classifiers designed to yield the same classification accuracy as a single-stage classifier (which forms the baseline) for all applications. We quantify efficiency in terms of three metrics: (i) average number of operations (or computations) per input (OPS), (ii) energy of hardware implementation, and (iii) energy of software implementation. We observe that scalable effort classifiers provide between 1.2×-9.8× (geometric mean: 2.79×) improvement in average OPS/input compared to the baseline. Note that the benefits vary depending on the fraction of hard-to-classify inputs in the dataset and the complexity of the classifier stages. For instance, the CONNECT application, in which we obtain the least improvement, filters only 25% of the inputs, while the complexity of the stages before the final stage add up to 10% of the original classifier. On the other extreme, the EYES application classifies 90% of the inputs at a cost of 0.2% of the baseline. In the case of hardware and software implementations, the reduction in OPS/input translates on an average to 2.3× and 1.5× improvement in energy, respectively. While still substantial, due to the control and memory overheads involved, the benefits in energy for some applications are lower than that in OPS/input. In particular, the impact of implementation overheads is pronounced in the case of applications with smaller feature sizes and datasets.

5.5.2 Impact of Hard Inputs on Efficiency

In this section, we examine the impact of hard-to-classify inputs on the overall efficiency of scalable-effort classifiers. Towards this end, we identify inputs that are

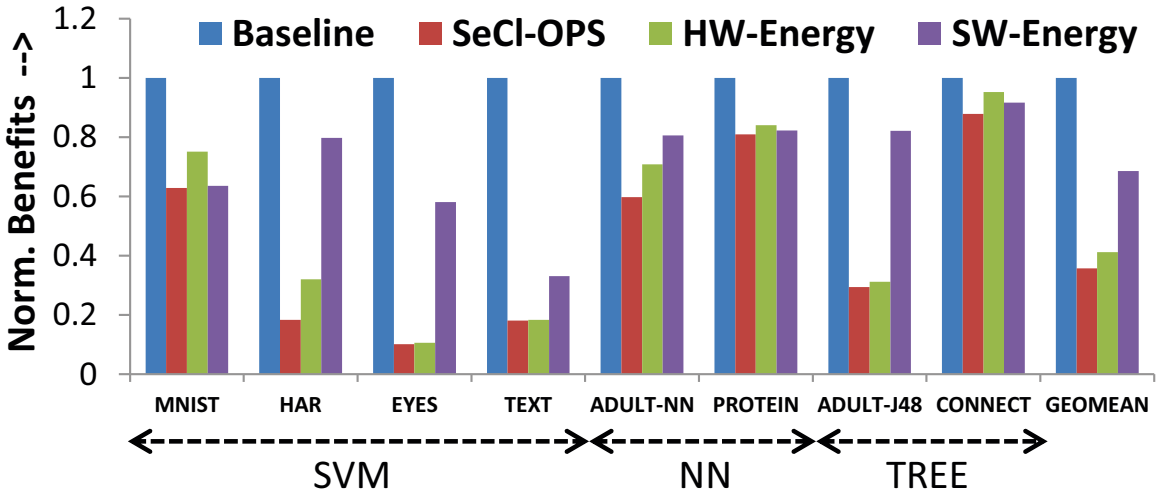


Fig. 5.7.: Improvement in average OPS/input and energy for different applications.

closer to the decision boundary of the original classifier and vary their proportion in the test dataset. Figure 5.8(a) shows the normalized OPS required for different fractions of hard inputs for three applications. Naturally, as the fraction of hard inputs increases, the benefits of scalable-effort execution are lowered. In fact, when the fraction increases beyond a certain level, scalable-effort classifiers become less efficient than the single-stage baseline depending on the application and the complexity of the additional classifier stages. Figure 5.8(b) shows the normalized complexities of the corresponding classifier stages. In the case of EYES, where the complexity of the added stages (all but the final stage) is only 0.2% of the single-stage classifier, scalable-effort design is desirable even when more than 99% of the inputs are hard [dashed vertical line in Figure 5.8(a)]. As the complexity of the added stages increases to 10% and 26%, as in the case of CONNECT and ADULT-NN, the break-even point occurs earlier at 85% and 72% of hard inputs, respectively. The observed fraction of hard inputs in these applications are also marked in Figure 5.8(a), which corresponds to the benefits reported in Sec. 5.5.1.

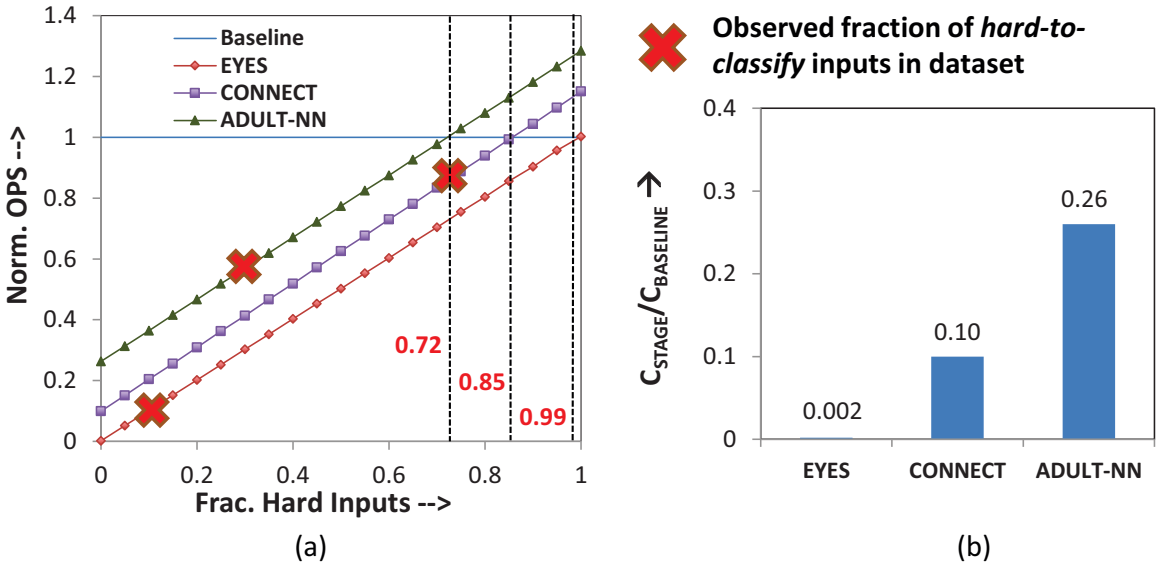


Fig. 5.8.: (a) Normalized OPS consumed by the scalable-effort classifier with increasing fraction of hard inputs. (b) Total complexity of the added classifier stages for different applications

5.5.3 Optimizing the Number of Classifier Stages

Choosing the right number of stages is critical to the efficiency of scalable-effort classifiers. We study the impact of this choice by varying the number of stages for the ADULT-J48 application. The normalized OPS of the overall classifier split amongst each stage is shown in Figure 5.9(a). When the number of stages is increased to 2, we see a large drop in total OPS, as the first stage adds only a small overhead, while significantly reducing the OPS contributed by the final stage. When we add a 3rd stage, we observe only a slight improvement. Although the stage decreases the number of final stage OPS, its added complexity nearly balances the reduction. Adding a 4th stage is unfavorable as it increases the overall OPS. To gain additional insight, consider the normalized stage complexity and the fraction of inputs classified at each stage shown for the 4-stage classifier in Figure 5.9(b). As expected, stage 1 is quite simple (5% complexity compared to single-stage classifier) and classifies a disproportionately large number (53%) of inputs. Stage 2 is balanced, with 24%

complexity and 29% classification rate. The trade-off is reversed for the third stage, whose complexity is 53%, but which classifies only 18% of additional inputs. This behavior is also reflected in the *gain* of each stage quantified by our design methodology in Figure 5.9(b).

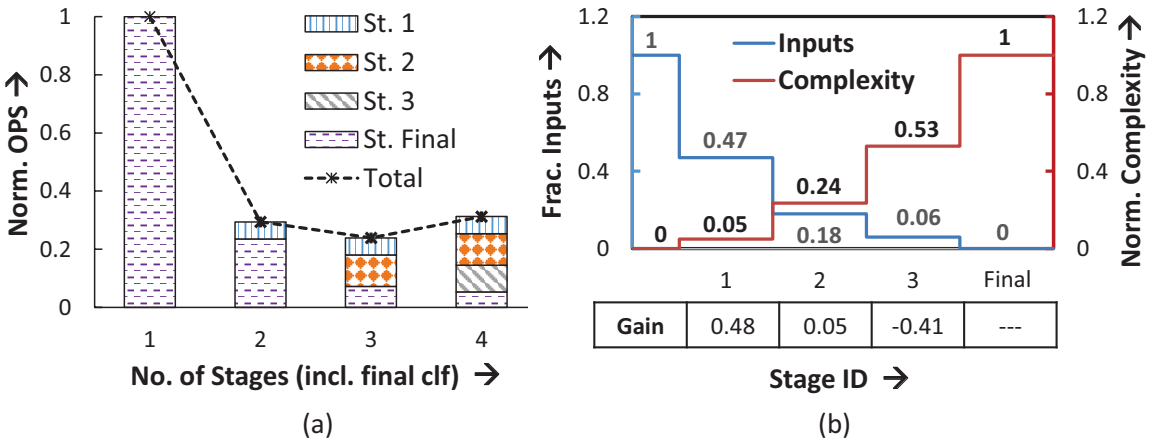


Fig. 5.9.: Normalized reduction in OPS with different numbers of classifier stages for the ADULT-J48 application

5.5.4 Efficiency-Accuracy Tradeoff using δ

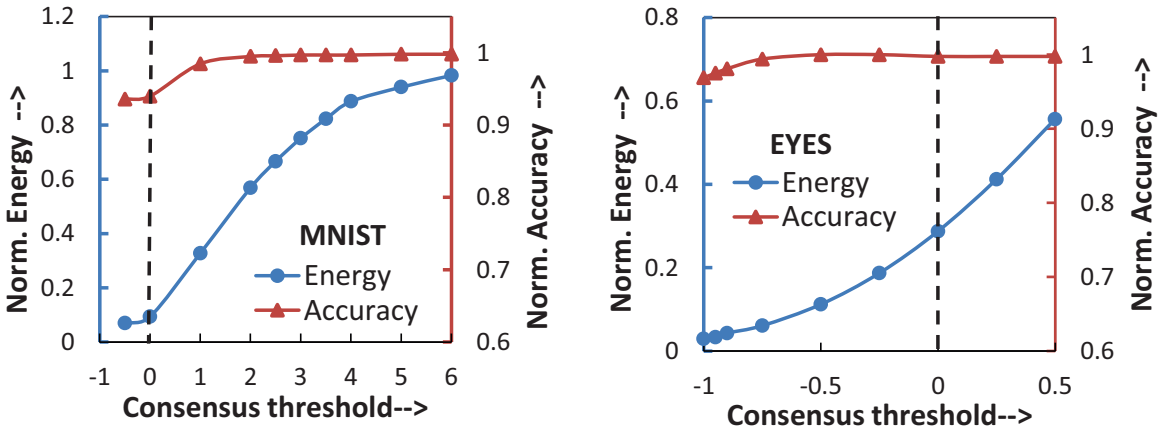


Fig. 5.10.: Energy *vs.* accuracy trade-off by modulating consensus threshold

The consensus threshold δ provides a powerful knob to trade accuracy for efficiency. Figure 5.10 shows the variation in the normalized energy and accuracy of the scalable-effort classifier with different values of δ for two applications. In the case of MNIST, when $\delta = 0$, the accuracy is $\sim 5\%$ lower than the baseline, with over $10\times$ improvement in energy. To reach the same level of accuracy, δ should be increased, but this comes at the cost of higher energy consumption since now more inputs will reach the final classifier stage. Decreasing δ improves efficiency, but further degrades accuracy. For the EYES application, we find that even when $\delta = 0$, the accuracy is on par with the original classifier and the energy is $3.5\times$ lower. If we lower δ to -0.5, the energy improvement increases to $9\times$ with minimal loss in accuracy. Further decreasing δ still leads to only a slight degradation in accuracy (3% lower accuracy for $30\times$ energy improvement for $\delta = -1$). Thus, the efficiency and accuracy of scalable-effort classifiers is a strong function of δ, which can be easily adjusted at runtime to an appropriate value.

5.6 Summary

Supervised machine learning algorithms play a central role in various important applications and place significant demand on the computational capabilities of modern computing platforms. In this thesis, we identify a new opportunity to optimize machine learning classifiers by exploiting the significant variability in the inherent classification difficulty of inputs. Based on the above insight, we propose the concept of scalable effort classifiers, or classifiers that dynamically scale their effort to match the complexity of the input being classified. We develop a systematic methodology to design scalable effort versions for any given classifier and training dataset. We achieve this by cascading biased versions of the classifier that progressively grow in complexity and accuracy. The scalable effort classifier is equipped to implicitly modulate the number of stages used for classification based on the input, thereby achieving scalable effort execution. To quantify the potential of scalable effort classifiers, we

build scalable effort versions of 8 recognition and computer vision applications that utilize 3 popular machine learning classifiers. Our experiments demonstrate 2.79× reduction in average OPS per input, which translates to 2.3× and 1.5× improvement in energy over hardware and software implementations of the applications.

6. RELATED WORK

A large body of previous work shares the philosophy of exploiting the intrinsic resilience of applications to achieve improvements in energy or performance. A wealth of approximate computing design techniques, spanning different levels of design abstraction, have been proposed towards this objective. We provide an overview of such efforts and illustrate the key aspects unique to the thesis.

6.1 Approximate Computing in Software

At the software level, techniques have been proposed to reduce the run-time complexity or parallel scalability of resilient applications. Towards this goal, techniques such as loop perforation [15], selective computation skipping [5], replacing expensive functions with cost effective approximate versions [16, 18], and dependency relaxation [17] have been proposed. All these efforts exploit error resilience through software techniques that are complementary to the techniques proposed in this thesis. In addition, several software analysis and profiling frameworks such as [4, 41–43, 84] to analyze the impact of approximations at the application-level and to verify if the application level output quality is satisfied have been proposed.

6.2 Hardware Design for Approximate Computing

One of the earliest proposals to advocate error resilience as a means for energy efficiency was ANT (Algorithmic noise tolerance) [9], which employed voltage overscaling along with algorithmic error control schemes for energy-efficient DSP. Similar design approaches have been proposed for image, video and multimedia applications [85–94]. The concept of utilizing the inherent resilience of multi-media applica-

tions to tolerate defects and improve fabrication yields was proposed in [6], and led to new approaches to manufacturing test [95–105]. Another effort, Significance driven computation (SDC) [12], proposed a design methodology in which computations are selectively approximated based on their significance in shaping the output quality. It illustrates the importance of confining errors to a subset of non-critical computations through a voltage-scalable motion estimator. The concept of probabilistic CMOS [10], wherein transistors and logic gates display a probabilistic rather than deterministic behavior was proposed as an energy-efficient alternative to always-correct models. This has led to a significant body of research on probabilistic and approximate computation [106–114]. More recently, scalable-effort design was proposed [7, 8, 115, 116] as a cross-layer approach to achieve improved energy benefits through approximate computing. Effort knobs capable of modulating energy expended by hardware were identified at design time, and co-optimized through a feedback control mechanism at runtime. While significant energy benefits have been demonstrated using the above design techniques, their applicability is restricted to design of application-specific hardware.

In the realm of programmable processors, approximate computing techniques have been investigated in [11,13,14,40]. Stochastic processors [11] and ERSA [40] consider multi-core architectures, in which individual processor cores differ in their reliability. They partition computations across reliable and unreliable cores at the granularity of tasks based on their criticality to output quality. On the other hand, [14] proposes to distinguish critical and non-critical instructions statically during compile time using programmer specified type annotations. Program instructions are then executed in an accurate or approximate manner with appropriate micro-architectural support [13]. All the above techniques carry a binary notion of quality with no guarantees from hardware on the error that it may incur during execution. As demonstrated in Figure 3.1, this significantly limits the extent to which approximate computing can be employed. Further, these techniques use lower voltage operation to realize approximations, which typically introduces errors of large magnitude since the most significant

bits (MSB) are timing critical and fail first under timing violations caused by supply voltage reduction. To address this limitation, we propose the notion of quality programmability in instructions in this research.

6.3 Approximate Circuits

Approximate circuits evaluate a given function with lower hardware complexity subject to an imposed quality constraint. A number of previous works have focused on manually approximating specific circuits like adders [19–21, 26, 117] and multipliers [23] by taking advantage of their structural properties and the difference in the significance of their output bits. However, all these design techniques are confined to the specific circuits that they target. Automation becomes necessary as circuits grow functionally complex and the approximations that can be performed on them become non-intuitive.

One class of automation techniques target synthesizing circuits that trade-off accuracy for power through voltage over-scaling. Traditional synthesis optimizations result in circuits that contain a large number of near-critical paths and impede aggressive voltage scaling. To ensure a graceful degradation in the number of timing violations under over-scaling, the path delay distribution of the circuit is reshaped by increasing the slack of frequently exercised paths through cell sizing [118] or common case promotion [119]. Techniques are proposed in [120, 121] to estimate and analyze the errors caused due to such approximations.

Improvement in power and performance could be alternatively achieved by simplifying the logic functions to reduce their implementation complexity. The first automation effort in this direction focused on two-level circuits, by complementing the output for selected minterms to reduce the sum-of-products implementation [122]. For multi-level circuits, [123] proposes a scheme where a node in the circuit is assumed to have a stuck-at-fault and the circuit is simplified by propagating this redundancy. The resultant errors are then estimated using simulation and a modified

automatic test pattern generation (ATPG) algorithm. This process is iterated until the pre-specified bounds are violated. A similar iterative approach is adopted in [124], however, pruning is instead carried out on paths with lowest path activation probabilities.

A common attribute of the above techniques is that they require design of a custom tool to perform the required approximations. In both cases, the quality metric is, in essence, hardwired into their synthesis procedures *i.e.,* the synthesis tools need to be substantially modified for using them with different error metrics. Also, these techniques do not perform any structural modifications to circuits but rather simplify circuits only through redundancy propagation or by pruning gates exclusive to a path. Lastly, these techniques mostly rely on simulations to testify if the approximate circuit adheres to the quality bounds.

In contrast, the proposed methodologies take a systematic approach to approximate logic synthesis. The problem is reformulated using circuit transformations and cast in such a manner that existing logic synthesis tools could be leveraged for approximate logic synthesis. This vastly enhances the extent of approximations applied, since the full suite of techniques used in logic synthesis tools can be utilized. In addition to pruning/removing gates, the approaches provide the capability to transform the functionality of circuit nodes. Also, they decouple the synthesis procedure from the target error metric, which make the approaches more generic and easily adaptable. Finally, they provide an inherent guarantee that the synthesized approximate circuit adheres to the pre-specified quality bounds. We believe that the above distinguishing traits make SALSA and SASIMI promising approaches to approximate and quality configurable circuit synthesis.

7. CONCLUSION

The *efficiency gap* created by diminishing benefits from semiconductor technology scaling on the one hand, and projected growth in computing and data demand on the other, has created an urgent need to identify new sources of computing efficiency across the computing stack. Fortunately, the workloads that drive the demand for computing efficiency also present new opportunities. In data centers and the cloud, the demand for computing is driven by the need to organize, search through, analyze, and draw inferences from, exploding amounts of digital data. In mobile and embedded devices, the need to more naturally and intelligently interact with the physical world, and process richer media drive much of the computing demand. A common pattern that emerges from both ends of the spectrum is that these applications are largely not about calculating a precise numerical answer; instead, "correctness" is defined as producing results that are good enough, or of sufficient quality, to produce an acceptable user experience. As a result, these workloads are endowed with a high degree of intrinsic resilience to their underlying computations being executed in an approximate or inexact manner.

Approximate computing broadly refers to exploiting the forgiving nature (or intrinsic resilience) of applications to design more efficient (faster, lower power) computing platforms. While prior efforts in approximate computing have established its potential for significant improvements, they have invariably been explored in an application-specific context – the techniques are often ad hoc and applicable to specific applications, or the end result is often application-specific custom hardware.

7.1 Thesis Summary

To establish approximate computing in a broader context, the thesis develops an holistic approach that includes automatic frameworks to synthesize approximate circuit blocks, a model for programmable approximate processors that explicitly codifies the notion of quality into the HW/SW interface, and finally software techniques to systematically identify resilient computations within an application and to apply approximate computing to achieve a favorable quality-efficiency tradeoff. The key contributions of the thesis are summarized below.

- The thesis proposed a rigorous circuit design framework for approximate computing. It developed two synthesis tools *viz.* SALSA and SASIMI, which given an original circuit and quality constraint can automatically generate approximate and quality configurable versions of the circuit. The tools were tested on a wide range of benchmarks to demonstrate their generality, scalability and efficiency.

- The thesis addressed the important problem building programmable approximate computing platforms. It proposed the concept of quality programmable processors, in which the notion of quality was codified as part of its HW/SW interface. The ISA was enhanced to facilitate software dictate the quality requirement of instructions and hardware was designed to understand and translate the flexibility into energy efficiency. These concepts were demonstrated for a range of applications using a quality programmable 1D/2D vector processor, Quora.

- Finally, in the context of machine/deep learning applications, the thesis proposed software framworks—AxNN and scalable effort classifiers—to systematically identify which computations are resilient to approximations. These frameworks leverage domain specific insights to achieve a superior efficiency *vs.* quality tradeoff. AxNN adapted backpropgation to charcterize criticality of neurons

in DLNs, and incrementally retrained the network with the approximations in place to partially heal their impact. On the other hand, scalable effort classfiers leveraged the inherent heterogentiy in the diffculty of inputs to machine learning classifiers to improve their efficiency. They comprised of a chain of classifiers of growing complexity (and accuracy), and the number of stages used for classification was dynamically modulated based on input difficulty. Such frameworks are critical to systematically map applications to approximate computing platforms.

7.2 Research Challenges

While many advances have been made in the area of approximate computing, much remains to be explored and many challenges need to be addressed. Some of the open research challenges in approximate computing are described below.

Quality specification, translation and verification. Since both intrinsic resilience and approximate computing arise from the notion of acceptable quality of results, it is important to have a clear, measurable definition of what constitutes acceptable quality. In addition, it is critical to develop methods to ensure that acceptable quality is maintained when approximate computing techniques are used. Broadly speaking, quality specification and verification remains an open challenge. It is important to note that quality metrics do vary across applications (recognition or classification accuracy, rele- vance of search results, visual quality of images or video, *etc.*). However, the abstractions and methodology used to specify and validate quality should still be general, and to the extent possible re-use tools and concepts from functional verification.

Identifying resilient computations. Given an abstraction to specify quality at the application-level, the next challenge is to identify which computations within the application can be subject to approximations and by how much. While the thesis systematically addresses this problem in the context of machine/deep learning

applications, a generic methodology applicable to any application remains an open challenge. This can be achieved in several ways. Profiling tools and auto-tuning frameworks with in-built approximation models can be developed. Another approach is to design quality configurable versions of common programming templates and libraries, which designers can directly utilize in their implementations.

Approximate general purpose scalar processors. The efficacy of approximate computing varies widely with the implementation context. For example, approximating algorithm-specific accelerators yield the most benefit as they have the least control overheads. This thesis proposed the notion of quality programmable processors and demonstrated it in the context of a vector processor. At the other end of the spectrum, general purpose scalar processors are challenging to approximate as control front-ends, such as instruction fetch and decode, inherent to any programmable processor, have to be performed in an accurate manner. In addition, the absolute number of control operations are also larger in a general purpose implementations. Therefore, approximate computing in their context should transcend beyond conventional numerical value-based approximations and explore how sequences of operations can be approximated together. ISA extensions to express such sequences need to be developed. Also, the approximations should target real bottlenecks to performance/energy to yield disproportionate benefits. This requires utilizing the dynamic information available during program execution. Thus, general purpose processors require a rethink of how approximate computing is realized and present interesting research opportunities.

Approximate memory and I/O subsystems A large majority of research in approximate computing has focussed on reducing the computation energy expended in the processing cores. However, the memory and I/O subsystems also constitute a significant fraction of the overall system energy, and their proportion is expected to grow at further scaled technologies. The principles of approximate computing can be applied to benefit energy in their context. Some preliminary research ideas in this direction include:

- *Quality-encoded data transfer*: There is a need to develop new re-configurable bus coding schemes that can encode data with different levels of information loss (and commensurate data-transfer energy) depending on explicit quality requirements. These quality constraints are derived based on the significance of the data in the context of the application and the operation that will be subsequently performed on it.

- *Quality-driven data sensing*: In many emerging applications, not all data sensed ends up being useful to the application. Techniques to sense and process data approximately near the sensors to gauge its relevance and utilize it to drive the quality of sensing are key to improve sensing efficiency. Such techniques will have significant impact in the context of distributed applications that continuously interact between multiple mobile/embedded end-points and the cloud.

- *Quality-aware data organization and access*: Most of the data created and stored today are unstructured. New approaches to re-organize data in memory at runtime based on the feedback about its significance from the application can render data querying and access more efficient.

In addition to the above research challenges, the integration of various approximate computing techniques proposed at differnt layers of the stack into a cohesive framework and evaluation of the benefits of approximate computing in real end systems are also critical to overcome designer mindset and the eventual adoption of approximate computing.

In summary, the thesis proposed an intergrated cross-layer framework to systematically design applications using approximate computig. The contributions of the thesis broaden the scope of approximate computing, thus promising a significant leap towards bringing approximate computing closer to the mainstream.

REFERENCES

REFERENCES

[1] G. E. Larson, R. J. Haier, L. LaCasse, and K. Hazen, "Evaluation of a "mental effort" hypothesis for correlations between cortical metabolism and intelligence," *Intelligence*, vol. 21, no. 3, pp. 267 – 278, 1995. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0160289695900179

[2] Y.-K. Chen, J. Chhugani, P. Dubey, C. Hughes, D. Kim, S. Kumar, V. Lee, A. Nguyen, and M. Smelyanskiy, "Convergence of recognition, mining, and synthesis workloads and its implications," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 790–807, 2008.

[3] P. Dubey, "A platform 2015 workload model recognition, mining and synthesis moves computers to the era of tera," White paper, Intel Corp., 2005.

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

[5] J. Meng, S. Chakradhar, and A. Raghunathan, "Best-effort parallel execution framework for recognition and mining applications," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009, pp. 1–12.

[6] M. Breuer, "Multi-media applications and imprecise computation," in *Digital System Design, 2005. Proceedings. 8th Euromicro Conference on*, 2005, pp. 2–7.

[7] V. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. Chakradhar, "Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency," in *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, 2010, pp. 555–560.

[8] V. Chippa, A. Raghunathan, K. Roy, and S. Chakradhar, "Dynamic effort scaling: Managing the quality-efficiency tradeoff," in *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, 2011, pp. 603–608.

[9] R. Hegde and N. Shanbhag, "Energy-efficient signal processing via algorithmic noise-tolerance," in *Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on*, 1999, pp. 30–35.

[10] K. V. Palem, L. N. Chakrapani, Z. M. Kedem, A. Lingamneni, and K. K. Muntimadugu, "Sustaining moore's law in embedded computing through probabilistic and approximate design: Retrospects and prospects," in *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '09. New York, NY, USA: ACM, 2009, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/1629395.1629397

[11] S. Narayanan, J. Sartori, R. Kumar, and D. Jones, "Scalable stochastic processors," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, 2010, pp. 335–338.

[12] D. Mohapatra, G. Karakonstantis, and K. Roy, "Significance driven computation: A voltage-scalable, variation-aware, quality-tuning motion estimator," in *Proceedings of the 14th ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '09. New York, NY, USA: ACM, 2009, pp. 195–200. [Online]. Available: http://doi.acm.org/10.1145/1594233.1594282

[13] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," *SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 301–312, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2189750.2151008

[14] 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. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993518

[15] 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. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025133

[16] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 198–209. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806620

[17] S. Byna, J. Meng, A. Raghunathan, S. Chakradhar, and S. Cadambi, "Best-effort semantic document search on gpus," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. New York, NY, USA: ACM, 2010, pp. 86–93. [Online]. Available: http://doi.acm.org/10.1145/1735688.1735705

[18] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, 2012, pp. 449–460.

[19] D. Shin and S. K. Gupta, "A re-design technique for datapath modules in error tolerant applications," in *Proc. ATS*, Nov. 2008, pp. 431 –437.

[20] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "Impact: Imprecise adders for low-power approximate computing," in *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, 2011, pp. 409–414.

[21] A. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, 2012, pp. 820–825.

[22] N. Olivieri, F. Pappalardo, S. Smorfa, and G. Visalli, "Analysis and implementation of a novel leading zero anticipation algorithm for floating-point arithmetic units," *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 54, no. 8, pp. 685–689, 2007.

[23] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *VLSI Design (VLSI Design), 2011 24th International Conference on*, 2011, pp. 346–351.

[24] D. Mohapatra, V. Chippa, A. Raghunathan, and K. Roy, "Design of voltage-scalable meta-functions for approximate computing," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, 2011, pp. 1–6.

[25] P. K. Krause and I. Polian, "Adaptive voltage over-scaling for resilient applications," in *Proc. DATE*, March 2011, pp. 1 –6.

[26] J. Miao, K. He, A. Gerstlauer, and M. Orshansky, "Modeling and synthesis of quality-energy optimal approximate adders," in *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, 2012, pp. 728–735.

[27] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*, 1st ed. McGraw-Hill Higher Education, 1994.

[28] H. Savoj and R. K. Brayton, "The use of observability and external don't cares for the simplification of multi-level networks," in *Proc. DAC*, 1990, pp. 297–301.

[29] K. H. Chang, V. Bertacco, I. L. Markov, and A. Mishchenko, "Logic synthesis and circuit customization using extensive external don't-cares," *ACM TODAES*, vol. 15, pp. 26:1–26:24, June 2010.

[30] S. Chang and M. M. Sadowska, "Perturb and simplify: optimizing circuits with external don't cares," in *Proc. ED TC*, mar 1996, pp. 402 –406.

[31] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "Salsa: Systematic logic synthesis of approximate circuits," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 796–801. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228504

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

[33] E. Sentovich and K. Singh, "SIS: A system for sequential circuit synthesis," EECS, UCB, Tech. Rep., 1992. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1992/2010.html

[34] "Design Compiler Ultra, Synopsys Inc."

[35] L. Benini, E. Macii, and M. Poncino, "Telescopic units: increasing the average throughput of pipelined designs by adaptive latency control," in *Proc. DAC*, 1997, pp. 22–27.

[36] D. Baneres, J. Cortadella, and M. Kishinevsky, "Variable-latency design by function speculation," in *Proc. DATE*, april 2009, pp. 1704 –1709.

[37] S. Ghosh, S. Bhunia, and K. Roy, "Crista: A new paradigm for low-power, variation-tolerant, and adaptive circuit synthesis using critical path isolation," *IEEE Trans. on CAD*, vol. 26, pp. 1947–1956, nov. 2007.

[38] S. L. Lu, "Speeding up processing with approximation circuits," *Computer*, vol. 37, no. 3, pp. 67 – 73, mar 2004.

[39] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[40] L. Leem, H. Cho, J. Bau, Q. Jacobson, and S. Mitra, "Ersa: Error resilient system architecture for probabilistic applications," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, 2010, pp. 1560–1565.

[41] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard, "Verified integrity properties for safe approximate program transformations," in *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM '13. New York, NY, USA: ACM, 2013, pp. 63–66. [Online]. Available: http://doi.acm.org/10.1145/2426890.2426901

[42] ——, "Proving acceptability properties of relaxed nondeterministic approximate programs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 169–180. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254086

[43] J. Cong and K. Gururaj, "Assuring application-level correctness against soft errors," in *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, 2011, pp. 150–157.

[44] "ModelSim, Mentor Graphics Inc."

[45] K. K. et. al, "Learning convolutional feature hierarchies for visual recognition," in *NIPS*, 2010.

[46] A. K. et. al., "Imagenet classification with deep convolutional neural networks," in *NIPS, 2012*.

[47] G. Rosenberg, "Improving photo search: A step across the semantic gap," June 2009.

[48] J. D. et. al., "Large scale distributed deep networks," in *NIPS*, 2012.

[49] "Scientists See Promise in Deep-Learning Programs, www.nytimes.com /2012/11/24/science/scientists-see-advances-in-deep-learning-a-part-of-artificial-intelligence.html."

[50] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556

[51] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," *CoRR*, vol. abs/1312.6229, 2013. [Online]. Available: http://arxiv.org/abs/1312.6229

[52] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[53] "Virgin atlantic introduces google glass in innovation drive to fuel the future of air travel: http://www.virgin-atlantic.com/in/en/footer/media-centre/press-releases/google-glass.html."

[54] "Fraunhofer IIS presents world's first emotion detection app on Google Glass: http://www.iis.fraunhofer.de/en/pr/2014/20140827_BS_Shore_Google_Glas.html."

[55] "Google glass teardown: http://www.catwig.com/google-glass-teardown/."

[56] "TI OMAP Datasheet:http://www.ti.com/product/omap3530."

[57] "Intel xeon phi coprocessor datasheet: http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-datasheet.html."

[58] C. F. et al., "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Proc. CVPRW*, 2011, pp. 109–116.

[59] S. C. et. al., "A dynamically configurable coprocessor for convolutional neural networks," in *Proc. ISCA*, 2010. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815993

[60] E. P. et. al., "Spinnaker: A multi-core system-on-chip for massively-parallel neural net simulation," in *Proc. CICC*, 2012, pp. 1–4.

[61] J. N. et. al., "On optimization methods for deep learning," in *Proc. ICML*, 2011, pp. 265–272.

[62] B. R. et. al., "Specifications of nanoscale devices and circuits for neuromorphic computational systems," *IEEE Trans. on Electron Devices*, vol. 60, no. 1, pp. 246–253, 2013.

[63] S. H. J. et. al., "Nanoscale memristor device as synapse in neuromorphic systems," *Nano Letters*, vol. 10, no. 4, pp. 1297–1301, 2010. [Online]. Available: http://pubs.acs.org/doi/abs/10.1021/nl904092h

[64] K. R. et al., "Beyond charge-based computation: Boolean and non-Boolean computing with spin torque devices," in *Proc. ISLPED*, Sep. 2013, pp. 139–142.

[65] O. Temam, "The rebirth of neural networks," in *Proc. ISCA*. [Online]. Available: pages.saclay.inria.fr/olivier.temam/homepage/ISCA2010web.pdf

[66] P. Dubey, "Recognition, mining and synthesis moves computers to the era of tera," *Intel Tech. Magazine*, vol. 9, no. 2, pp. 1–10, Feb. 2005.

[67] L. Deng, "The MNIST database of handwritten digit images for machine learning research," *IEEE Signal Proc. Magazine*, vol. 29, no. 6, pp. 141–142, Nov. 2012.

[68] Y.-T. L. *et al.*, "Low-power variable-length fast fourier transform processor," *Proc. IEEE Computers and Digital Techniques*, vol. 152, no. 4, pp. 499–506, Jul. 2005.

[69] H. K. *et al.*, "A 1.45 GHz 52-to-162 GFLOPS/W variable-precision floating-point fused multiply-add unit with certainty tracking in 32nm CMOS," in *Proc. ISSCC*, Feb. 2012, pp. 182–184.

[70] V. C. *et al.*, "Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency," in *Proc. DAC*, 2010, pp. 555 –560.

[71] ——, "Dynamic effort scaling: Managing the quality-efficiency tradeoff," in *Proc. DAC*, June 2011, pp. 603–608.

[72] H. E. *et al.*, "Architecture support for disciplined approximate programming," in *Proc. Int. Conf Architectural Support for Programming Languages and Operating Systems*, Mar. 2012, pp. 301–312.

[73] S. S.-D. et. al., "Managing performance vs. accuracy trade-offs with loop perforation," in *Proc. ACM SIGSOFT Symposium*, 2011, pp. 124–134.

[74] S. N. *et al.*, "Scalable stochastic processors," in *Proc. Design Automation and Test in Europe*, 2010, pp. 335–338.

[75] S. V. et al., "Quality programmable vector processors for approximate computing," in *Proc. MICRO*, 2013, pp. 1–12.

[76] R. E. Schapire, "The boosting approach to machine learning: An overview," *Lect. Notes in Statistics: Nonlinear Estim. and Classification*, vol. 171, pp. 149–171, 2003.

[77] J. Gama and P. Brazdil, "Cascade generalization," *J. Machine Learning*, vol. 41, no. 3, pp. 315–343, Dec. 2000.

[78] P. V. *et al.*, "Rapid object detection using a boosted cascade of simple features," in *Proc. Conf. Comput. Vision & Pattern Reco.*, Dec. 2001, pp. 511–518.

[79] D. H. *et al.*, "Accelerating Viola-Jones face detection to FPGA-level using GPUs," in *Symp. Field-Programmable Custom Computing Machines*, May. 2010, pp. 11–18.

[80] C. Zhang and P. Viola, "Multiple-instance pruning for learning efficient cascade detectors," in *Proc. Neural Info. Processing Syst.*, Dec. 2008, pp. 1681–1888.

[81] R. Quinlan, *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann Publishers, 1993.

[82] K. Bache and M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[83] M. H. *et al.*, "The WEKA data mining software: an update," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, Jun. 2009.

[84] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 33–52. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509546

[85] N. Shanbhag, "Reliable and energy-efficient digital signal processing," in *Proc. Design Automation Conference*, 2002, pp. 830–835.

[86] R. Hegde and N. Shanbhag, "A low-power digital filter IC via soft DSP," in *Proc. IEEE Conf. Custom Integrated Circuits*, 2001, pp. 309–312.

[87] G. V. Varatkar and N. R. Shanbhag, "Error-resilient motion estimation architecture," *IEEE Trans. VLSI Systems*, vol. 16, no. 10, pp. 1399–1412, 2008.

[88] L. Wang and N. R. Shanbhag, "Noise-tolerant dynamic circuit design," in *ISCAS (1)*, 1999, pp. 549–552.

[89] R. Hegde and N. R. Shanbhag, "Toward achieving energy efficiency in presence of deep submicron noise," *IEEE Trans. VLSI Syst.*, vol. 8, no. 4, pp. 379–391, 2000.

[90] N. R. Shanbhag, K. Soumyanath, and S. Martin, "Reliable low-power design in the presence of deep submicron noise (embedded tutorial session)," in *ISLPED*, 2000, pp. 295–302.

[91] L. Wang and N. R. Shanbhag, "Energy-efficiency bounds for deep submicron vlsi systems in the presence of noise," *IEEE Trans. VLSI Syst.*, vol. 11, no. 2, pp. 254–269, 2003.

[92] B. Shim and N. R. Shanbhag, "Performance analysis of algorithmic noise-tolerance techniques," in *ISCAS (4)*, 2003, pp. 113–116.

[93] M. Zhang and N. R. Shanbhag, "An energy-efficient circuit technique for single event transient noise-tolerance," in *ISCAS (1)*, 2005, pp. 636–639.

[94] R. Hegde and N. R. Shanbhag, "Energy-efficiency in presence of deep submicron noise," in *ICCAD*, 1998, pp. 228–234.

[95] T.-Y. Hsieh, K.-J. Lee, and M. Breuer, "An error rate based test methodology to support error-tolerance," *Reliability, IEEE Transactions on*, vol. 57, no. 1, pp. 204–214, March 2008.

[96] Z. Pan and M. A. Breuer, "Basing acceptable error-tolerant performance on significance-based error-rate (sber)," in *VTS*, 2008, pp. 59–66.

[97] M. A. Breuer and H. H. Zhu, "An illustrated methodology for analysis of error tolerance," *IEEE Design & Test of Computers*, vol. 25, no. 2, pp. 168–177, 2008.

[98] T.-Y. Hsieh, K.-J. Lee, and M. A. Breuer, "An error rate based test methodology to support error-tolerance," *IEEE Transactions on Reliability*, vol. 57, no. 1, pp. 204–214, 2008.

[99] M. A. Breuer and H. H. Zhu, "Error-tolerance and multi-media," in *IIH-MSP*, 2006, pp. 521–524.

[100] *Second International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP 2006), Pasadena, California, USA, December 18-20, 2006, Proceedings*. IEEE Computer Society, 2006.

[101] T.-Y. Hsieh, K.-J. Lee, and M. A. Breuer, "An error-oriented test methodology to improve yield with error-tolerance," in *VTS*, 2006, pp. 130–135.

[102] M. A. Breuer, "Multi-media applications and imprecise computation," in *DSD*, 2005, pp. 2–7.

[103] ——, "Intelligible test techniques to support error-tolerance," in *Asian Test Symposium*, 2004, pp. 386–393.

[104] M. A. Breuer, S. K. Gupta, and T. M. Mak, "Defect and error tolerance in the presence of massive numbers of defects," *IEEE Design & Test of Computers*, vol. 21, no. 3, pp. 216–227, 2004.

[105] M. A. Breuer, "Determining error rate in error tolerant vlsi chips," in *DELTA*, 2004, pp. 321–326.

[106] K. V. Palem, "Computational proof as experiment: Probabilistic algorithms from a thermodynamic perspective," in *Verification: Theory and Practice*, 2003, pp. 524–547.

[107] ——, "Energy aware algorithm design via probabilistic computing: from algorithms and models to moore's law and novel (semiconductor) devices," in *CASES*, 2003, pp. 113–116.

[108] ——, "Energy aware computing through probabilistic switching: A study of limits," *IEEE Trans. Computers*, vol. 54, no. 9, pp. 1123–1137, 2005.

[109] P. Korkmaz, B. E. S. Akgul, and K. V. Palem, "Ultra-low energy computing with noise: Energy-performance-probability trade-offs," in *ISVLSI*, 2006, pp. 349–354.

[110] J. George, B. Marr, B. E. S. Akgul, and K. V. Palem, "Probabilistic arithmetic and energy efficient embedded signal processing," in *CASES*, 2006, pp. 158–168.

[111] L. N. Chakrapani, P. Korkmaz, B. E. S. Akgul, and K. V. Palem, "Probabilistic system-on-a-chip architectures," *ACM Trans. Design Autom. Electr. Syst.*, vol. 12, no. 3, 2007.

[112] B. E. S. Akgul, L. N. Chakrapani, P. Korkmaz, and K. V. Palem, "Probabilistic cmos technology: A survey and future directions," in *VLSI-SoC*, 2006, pp. 1–6.

[113] L. N. Chakrapani, B. E. S. Akgul, S. Cheemalavagu, P. Korkmaz, K. V. Palem, and B. Seshasayee, "Ultra-efficient (embedded) soc architectures based on probabilistic cmos (pcmos) technology," in *DATE*, 2006, pp. 1110–1115.

[114] L. N. Chakrapani, K. K. Muntimadugu, L. Avinash, J. George, and K. V. Palem, "Highly energy and performance efficient embedded computing through approximately correct arithmetic: a mathematical foundation and preliminary experimental validation," in *CASES*, 2008, pp. 187–196.

[115] V. Chippa, H. Jayakumar, D. Mohapatra, K. Roy, and A. Raghunathan, "Energy-efficient recognition and mining processor using scalable effort design," in *Custom Integrated Circuits Conference (CICC), 2013 IEEE*, 2013, pp. 1–4.

[116] V. K. Chippa, K. Roy, S. T. Chakradhar, and A. Raghunathan, "Managing the quality vs. efficiency trade-off using dynamic effort scaling," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 2s, pp. 90:1–90:23, May 2013. [Online]. Available: http://doi.acm.org/10.1145/2465787.2465792

[117] 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," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 18, no. 8, pp. 1225–1229, 2010.

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

[119] L. Wan and D. Chen, "Ccp: Common case promotion for improved timing error resilience with energy efficiency," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '12. New York, NY, USA: ACM, 2012, pp. 135–140. [Online]. Available: http://doi.acm.org/10.1145/2333660.2333695

[120] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan, "Macaco: Modeling and analysis of circuits for approximate computing," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '11. Piscataway, NJ, USA: IEEE Press, 2011, pp. 667–673. [Online]. Available: http://dl.acm.org/citation.cfm?id=2132325.2132474

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

[122] D. Shin and S. K. Gupta, "Approximate logic synthesis for error tolerant applications," in *Proc. DATE*, Mar. 2010, pp. 957–960.

[123] D. Shin and S. Gupta, "Approximate logic synthesis for error tolerant applications," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, 2010, pp. 957–960.

[124] A. Lingamneni, C. Enz, J. L. Nagel, K. Palem, and C. Piguet, "Energy parsimonious circuit design through probabilistic pruning," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, 2011, pp. 1–6.

VITA

VITA

Swagath Venkataramani received the bachelors degree in Electrical and Electronics Engineering from the College of Engineering, Anna University, Guindy, India, in 2010, as the University Gold Medalist. He is currently pursuing the Ph.D. degree with the School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA. After his graduation, Swagath will join IBM T.J. Watson Research Center as a research staff member in the system architecture and design department.

Previously, Swagath was a visiting research scientist with Parallel Computing Labs, Intel, Bangalore, India. He has also been with the Exa-Scale Computing Group, Intel, Hillsboro, OR, USA, as part of the U.S. DOE's FastForward Program, and with the Sensing and Energy Research Group, Microsoft Research, Seattle, WA, USA. His current research interests include approximate computing, energy-efficient machine/deep learning, heterogeneous parallel architectures, computing with spintronic devices, and computational imaging.

Swagath's dissertation research was awarded the Intel Ph.D. Fellowship in computing leadership and the Purdue Bilsland Dissertation Fellowship. His research has received two best paper nominations from DAC 2016 and ISLPED 2014, a best-in-session award from TECHON 2016, and was adjudged the winner of ACM SIGDA DAC PhD Forum 2016. It has also been featured in MIT Technology Review, Slashdot, Physics Today, and NSF News From the Field.