PADERBORN UNIVERSITY

DISSERTATION

MCTS-based Approximate Accelerator Synthesis

by
Muhammad Awais

A thesis submitted in fulfillment of the requirements for the degree of Dr. rer. nat.

in the

Faculty for Computer Science, Electrical Engineering and Mathematics

June 29, 2021

Dedicated to my parents and my family

Acknowledgements

I would like to express my gratitude towards my supervisor Prof. Dr. Marco Platzner. I had an excellent time under his supervision with worthy and fruitful discussions that always turned out to be useful ideas. He would always see challenges as opportunities and tries to uncover new dimensions to fuel ones thoughts. I always recall meetings with him as a source of motivation and confidence boosting. I must acknowledge his kindness and an always positive attitude towards his students. Further, I would like to thank:

- Prof. Dr. Sybille Hellebrand for being reviewer of the thesis.
- All the committee members for the thesis evaluation.
- Dr. Hassan Ghasemzadeh Mohammadi for his contribution as a post-doc guide in this dissertation. The discussion and brainstorming sessions with him helped me and my thoughts to converge resulting in effective research ideas.
- My office-mate Tobias Wiersema with whom I had great discussions and tremendous memories of working together.
- My colleagues Qazi Arbab Ahmed, Alexander Boschmann, Lennart Clausing, Tobias Graf, Tim Hansmeier, Nam Ho, Paul Kaufmann, Christian Lienen, Achim Lösch, Sebastian Meisner, Jennifer Neumann, Antoniou Paraskewi, Linus Witschen, Tobias Kenter, Michael Lass, Heinrich Riebler, Prof. Dr. Christian Plessl, and Gavin Vaz for a nice professional working environment and valuable discussions over the time.

I acknowledge the high-performance compute resources provided by the *Paderborn Center of Parallel Computing* (PC^2) to run my computationally extensive experiments.

I would also like to thank the Higher Education Commission of Pakistan, DAAD Germany and Committee for Research and Junior Academics of Paderborn Unversity for providing me the opportunity to complete my work through their support.

Abstract

Since the beginning of this century, a slowdown of the technology scaling trend steered the focus of the researchers to investigate alternative avenues to keep the performance improvement trend continue. In addition to a seemingly long-term goal to find new materials and devices to replace conventional CMOS chips, multi-core and many-core chips in the meantime have become popular alternatives to improve throughput through parallel processing. Even though these architectures offer massive parallelism, writing code for such architectures is not straight forward as the algorithms need to be modified in accordance with the underlying architecture.

Another important concern that rose during the last two decades is the size and complexity of data. Applications today are required to execute complex operations on huge amounts of data. This results in increased resource utilization, e.g., higher energy consumptions and greater hardware cost.

Fortunately, a wide variety of applications are intrinsically resilient to the error, i.e., they can produce acceptable results despite errors in the underlying computations. This has led to the emergence of approximate computing; a new computing paradigm that can leverage error resilience to generate faster and more energy or area efficient solutions.

Automated approximate accelerator synthesis aims for one or more approximate accelerator instances that offer substantial energy and/or area savings while satisfying quality requirements. Search-based or analytical methods have been commonly utilized to generate approximate accelerators whereas the former offers larger set of possibilities for approximation and is time-costly, the latter however has shorter runtime but is not as flexible. Moreover, search-based methods mostly face combinatorial explosion due to exponential growth of the nodes in the search tree. Existing works apply heuristic methods that are greatly prone to overlook the paths leading to the global optimum.

This thesis proposes novel methods for the automated synthesis of approximate accelerators together with efficient pruning of the design space to allow better exploration of the approximate solutions. An additional contribution is to overcome the above-mentioned limitations of both analytical and search-based techniques. The works presented in the various chapters of this thesis explain in detail how these goals were targeted. First, an automated framework based on Monte Carlo Tree Search (MCTS) for efficient design space exploration together with a pruning strategy is developed. Later, the MCTS-based design space exploration is further enhanced by combining it with an analytical approximation phase and enabling

parallel exploration of the search space altogether providing an effective tool for automated synthesis of approximate accelerators. Finally, fast and accurate deep learning-based error estimation models are developed to speed up the MCTS-based synthesis framework. Throughout the thesis, the work is evaluated on practical benchmark accelerator circuits selected from various domains of applications and compared with state-of-the-art works to demonstrate the effectiveness of the proposed framework.

Zusammenfassung

Seit Beginn des Jahrhunderts hat die Verlangsamung des Verkleinerungstrends bei Halbleiterbauelementen dazu geführt, dass der Fokus von Wissenschaftlern zunehmend auf alternative Ansätze zur Steigerung der Rechenleistung gelenkt wurde. Zusätzlich zu dem offensichtlich langfristigen Ziel, durch neue Materialien und Fertigungstechniken konventionelle CMOS Chips zu ersetzen, sind Mehr- und Vielkern Chips in der Zwischenzeit zu beliebten Alternativen geworden, um den Durchsatz mit paralleler Verarbeitung zu erhöhen. Aufgrund der massiven Parallelität ist es häufig allerdings aufwändig , Code für solche Rechnerarchitekturen zu schreiben, da die Algorithmen an die zugrundeliegende Architektur angepasst werden müssen.

Ein weiterer wichtiger Aspekt, der in den letzten zwei Jahrzehnten an Bedeutung gewonnen hat, ist die Menge und Komplexität der Daten. Heutige Anwendungen müssen komplexe Operationen auf großen Datenmengen ausführen. Dies führt zu einem erhöhten Ressourcenverbrauch, z.B. in Bezug auf den Energieverbrauch oder die Hardwarekosten.

Glücklicherweise besitzen viele Anwendungen eine intrinsische Robustheit gegenüber Rechenfehlern, d.h. sie können auch unter Verwendung von fehlerbehafteten Berechnungen akzeptable Ergebnisse liefern. Dies hat zur Entstehung des Forschungsbereichs Approximate Computing geführt – ein neues Paradigma für Computerarchitekturen, welches die Fehlerresilienz gezielt ausnutzt, um schnellere oder energieeffizientere Schaltungen zu generieren.

Die automatisierte Synthese approximativer Beschleuniger zielt darauf ab, approximative Beschleuniger mit substanziellen Energie- und/oder Flächeneinsparungen zu generieren, unter Einhaltung bestimmter Qualitätsanforderungen. Üblicherweise werden zur Synthese approximativer Beschleuniger suchbasierte oder analytische Methoden angewandt. Erstere bieten eine größere Palette von Möglichkeiten zur Approximation und sind zeitaufwändig, während letztere zwar eine geringere Laufzeit aufweisen, aber dafür auch weniger flexibel sind. Zudem sind suchbasierte Methoden aufgrund des exponentiellen Wachstums der Knoten im Suchbaum häufig von kombinatorischer Explosion betroffen. Bereits existierende Arbeiten wenden heuristische Methoden an, welche allerdings sehr anfällig dafür sind, die Pfade zum globalen Maximum zu übersehen.

Diese Dissertation präsentiert neuartige Methoden zur automatisierten Synthese approximativer Beschleuniger in Kombination mit effizienter Suchraumbeschneidung, um eine verbesserte Erkundung der approximierten Lösungen zu erlauben.

Ein weiterer Aspekt dieser Arbeit ist die Überwindung der oben genannten Limitierungen analytischer und suchbasierter Methoden. Die Details dieser Beiträge werden in den verschiedenen Kapiteln der Dissertation erläutert. Zuerst wird ein automatisiertes Framework entwickelt, welches Monte Carlo Tree Search (MCTS) mit einer Beschneidungsstrategie für eine effiziente Erkundung des Suchraums kombiniert. Die MCTS-basierte Erkundung des Suchraums wird anschließend durch Vorschaltung einer analytischen Approximationsphase, sowie der parallelen Erkundung des Suchraums weiter verbessert, was zusammen ein leistungsfähiges Werkzeug für die automatisierte Synthese approximativer Beschleuniger darstellt. Abschließend werden schnelle und genaue Fehlerschätzmodelle präsentiert, welche auf Deep-Learning Verfahren basieren und genutzt werden, um das MCTS-basierte Synthese-Framework zu beschleunigen. Die in dieser Dissertation vorgestellten Ansätze werden durchgängig auf praxisnahen Benchmark-Schaltungen aus verschiedenen Anwendungsbereichen evaluiert und mit alternativen Ansätze auf dem aktuellen Stand der Technik verglichen, um die Leistungsfähigkeit des entwickelten Frameworks zu demonstrieren.

Table of Contents

Chapter 1

Introduction

This chapter begins by explaining the aftermath of the slow-down of technology scaling trend and the ever-increasing demand for computational resources in modern day computing. It then motivates the reader towards the use of approximate computing that can provide tremendous improvements in energy consumption and silicon footprints for a wide range of error-tolerant applications. Finally, the organization of the rest of the thesis is outlined.

1.1 Challenges for CMOS technology in the Nano-era

The continuous increase in the clock frequencies of commercial processor chips remained a major driving force for performance improvements until the beginning of this century. This trend has started to slow down due the physical device limitations that resists the continuation of the correlated trend of power and clock frequency as the size of transistor shrinks. The slow-down has urged researchers to look for other alternatives for performance improvements. These efforts have been expanding in multiple directions such as looking for new architectures, finding new materials that can replace conventional Complementary Metal Oxide Semiconductor (CMOS)-based chips, and mix of these two approaches. Figure 1.1 shows the technology scaling trends originating from the current general purpose CMOS-based chips (two bubbles near the origin of the graph) [1]. Three different directions of research expanding along three axes are shown in Figure 1.1 with names of specific architectures and devices shown inside the bubbles. Along the x-axis, the bubbles represent different new architectures and packaging schemes to harvest more performance e.g., system on chips and reconfigurable computing. Along the y-axis, different new devices and materials in the bubbles represent various endeavors to replace traditional CMOS-based chips. The bubbles along z-axis however, show different attempts that are in some way a combination of new devices and architectures to enable gains in performance.

Although the line of research looking for new materials has shown promising results and many devices / materials have been experimentally shown to have great potential, yet there is no single device that can be named as an obvious successor of the conventional CMOS-based chips [1].

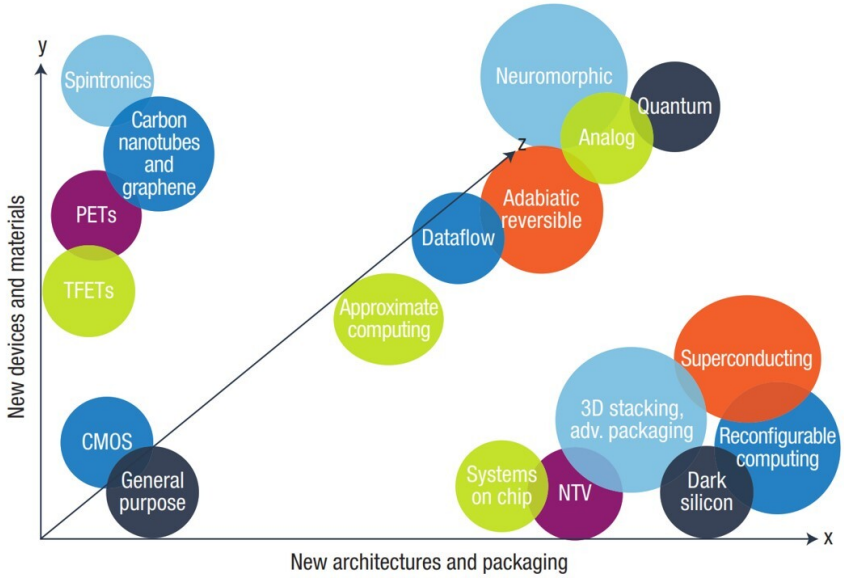
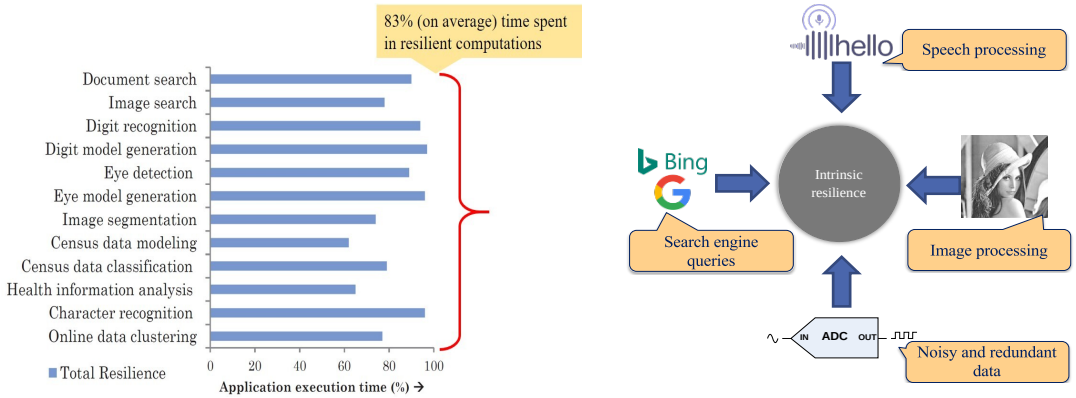


Figure 1.1: Technology scaling options along three dimensions [1].

In the meantime, other lines of research have attempted to improve performance via the parallel architectures; the multi-core and many-core architectures (Graphic Processing Units (GPUs), etc.). Although these architectures provide massive parallelism, programming them is not straight forward since a programmer need to have an in-depth understanding of the underlying architecture to be able to fully exploit the parallelism offered by these architectures. Higher power dissipation of such architectures is another concern specially for large organizations and data centers that provides continuous services to the users. In fact, it has been revealed in a couple of recent papers [2, 3], that a large portion of the silicon chip area in such architectures cannot be utilized simultaneously due to so-called dark silicon effect. Furthermore, for a full and fair utilization of heterogeneous architectures consisting of a Central Processing Unit (CPU), GPU(s) and Field Programmable Gate Array (FPGA) device(s), extensive expertise are required to develop scheduling algorithms for job allocations to available resources (architectures).

Besides the limitations of transistor scaling, another performance-limiting factor is the complexity of the tasks performed by the digital computers. Due to growth of the data that has to be processed for many complex tasks such as object recognition and classification, social networking websites, virtual reality and data warehouses, there is an ever-growing need of performance improvement. In addition to that, large data centers holding massive amount of information also consume more energy each year as the amount of data and tasks performed on the data increases [4].



(a) Resilience profile of various applications [5] (b) Different sources of resilience [6]

Figure 1.2: Error-resilient applications and various sources of resilience.

1.2 Approximate computing

Approximate Computing (AC) is a new computing paradigm that has recently gained much attention as an alternative approach for performance improvements and reduction in energy consumption and silicon footprints for a wide range of applications. AC exploits the fact that several applications are error resilient i.e., they can still produce results with acceptable precision if the underlying computations were performed in an imprecise manner. Recently, for a set of 12 commonly used applications from classification, recognition, regression and data mining categories, it was revealed in [5] that 83% (on average) of the total time was spent in error resilient parts. Figure 1.2-a lists the applications and their respective resilience percentages.

The resilience might come from multiple sources. Some examples are shown in Figure 1.2-b. A major source of resilience comes from the noisy and redundant data that many applications might receive as input from sources such as sensors. Another noticeable source is the fact that many applications do not have a golden result to be used as a reference and therefore there can be many solutions satisfying the users demand and any of them can be acceptable to the user e.g., the results returned by a recommender system or a search query. Limited human perception can also cause some imprecision go unnoticed. An example is an image returned as output by an image processing application where each pixel value slightly diverts from the golden value but still the visual appearance has no visible artifacts to be noticed.

An example is shown in Figure 1.3 to demonstrate why approximate computing makes sense. Here, a popular image sharpening algorithm (taken from [7]) is used to increase the intensity of the input image (shown in Figure 1.3-a). Figure 1.3-b shows the result obtained by applying the image sharpening filter using the exact operations where as the in Figure 1.3-c, the result of the same algorithm is shown with a large proportion of operations replaced by the approximate counterparts (the approximate version of the image sharpening accelerator circuit was generated by our automated synthesis flow explained in the Chapter 3). Although the visual impact

(a) Original (b) Sharpened (exact) (c) Sharpened (approximate), PSNR=49.72 dB

Figure 1.3: Comparison of exact and approximate image sharpening.

of the approximation is hard to notice, the approximate version takes 67% of the area as compared to the exact implementation when synthesized for the hardware. The quality of the approximate image is shown with the *Peak Signal-to-Noise Ratio* (PSNR) which is one of the most common metrics to represent the quality of the images. For image processing applications, higher values of PSNR are associated with higher quality (typically $PSNR > 30dB$ is considered good quality [4]). The PSNR value of approximate image in Figure 1.3-c is 49.72 dB which is well above acceptable quality.

1.2.1 Approximate computing at different layers of system stack

Recent years have witnessed an increasing amount of work in the field of approximate computing that ranges from application-level approximations to low-level transistor logic circuits [4]. Figure 1.4 outlines different approximate computing approaches applied at different abstraction layers of computing system stack. Some of the approaches can be applied at multiple levels (such as precision scaling). In the following, a brief review is presented that summarizes various works applying approximate computing techniques at different levels of computing system stack.

Application- / Algorithm-level approximate computing techniques mostly deal with high-level source code transformations. An example is loop perforation [8] that skips iterations in order to reduce the computations in a loop resulting in overall reduction in the energy consumed by the application. Other examples include relaxing synchronization and skipping tasks in a multi-core architecture [9], and identifying and reducing precision of the less contributing neurons in the backpropagation step of a deep neural network [10].

Language- / Compiler-level works develop programming language extensions enabling programmers to select approximate parts in the code. An example is *EnerJ* programming language [11] in which a programmer can select operations via code annotations that should be mapped to energy efficient hardware. In another work, *Axilog* [12] language is proposed that lets a programmer select the operations for

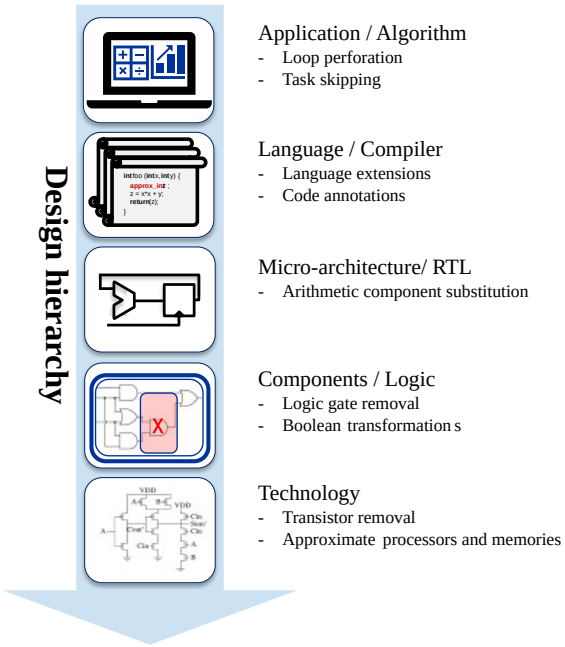


Figure 1.4: Approximate computing at different layers of computing stack.

which the accuracy requirements may be relaxed. This information is then utilized by the synthesis phase to approximate the parts of logic that are used in computing the operation. A compiler-driven error analysis and approximation flow is proposed in [13] that relies again on code annotations to find instructions in the high-level code which are then substituted with their approximate counterparts.

Micro-architecture- / RTL-level works mainly employ arithmetic component substitution as their approximation strategy. In arithmetic component substitution, parts of original circuit of an accelerator that are amenable to approximation are identified (often called candidates) and are then replaced with approximate arithmetic modules in a controlled and systematic way [14, 15, 16, 17, 18]. Various techniques have been proposed in the literature to obtain the approximate arithmetic modules such as truncating least significant bits, cutting carry chain, etc. Some open-source libraries provide wide range of approximate arithmetic modules with characterization of error and target parameters. A large number of available choices for these modules also uncovers challenges because the design space becomes huge and selection of components from the library that optimizes the target metric becomes non-trivial. A brief review of the approximate arithmetic modules follows in the next subsection.

A number of systematic methods have been introduced for automated generation of approximate accelerators from a given original accelerator circuit [14, 15, 16, 17, 18]. Such methods can be categorized on the basis of certain characteristics such as how they apply approximations or how the design space is explored. One such classification along with detailed discussion is presented in Section 2.2.

Components- / Logic-level works employ logic simplification methods for circuits at the gate/netlist level. Typically, first the parts of circuit logic that do not contribute much to the output are identified and then either those parts are truncated from the circuit or the nodes in those parts are connected to constant values to let the subsequent synthesis step simplify the circuit [19, 20]. For instance, in [19], the largest part of circuit logic is first identified by traversing its labeled direct acyclic graph representation via an exhaustive search such that removing that part will not violate the error bounds. The identified part of the logic is then removed from the circuit and the remaining logic is presented as the approximate circuit. Other examples are signal substitution [21], gate-level pruning [20], Boolean matrix factorization-based approximations [22], and random gate replacement [23].

Technology-level methods include several ad hoc techniques to create approximate circuits or systems. Broadly speaking, these works can be categorized as techniques that use manual circuit alteration to create approximate circuits at a lower level of abstraction, and techniques that are employed to generate approximate custom hardware architectures. Examples of former group of techniques include manual transistor removal from the transistor-level logic circuit of a gate and then use it to generate approximate half/full adders [24] or replacing part of adder circuit with OR gates [25]. These smaller adder modules are then combined to form bigger adder units and utilized in an image processing application. In other works such as in [26], approximate multipliers are generated via manipulation of a few entries of $K - map$ in a 2×2 multiplier and then this multiplier is used as a building block of a larger multiplier unit.

Many works have focused on the generation of custom hardware architecture that includes specialized CPU architectures, approximate memories, and caches to support approximate computations. A configurable processor with an extended instruction set architecture is proposed in [27]. The processor contains a hierarchical arrangement of processing elements that can monitor quality and perform dynamic precision scaling to yield different quality-energy trade-offs. Specialized approximate accelerators for emerging architectures such as coarse-grained reconfigurable architectures and in-memory computing architectures [28, 29] have also been targeted. Some other works exploit memory units for approximation such as approximate caches based on load-value approximation for traditional caches architecture [30] and for new technologies [31]. For traditional memory units, there has been DRAMs based on reduced refresh rates [32] and SRAMs based on reduced supply voltage [33]. Another example is the use of neural networks to achieve code acceleration [34]. It works by off-loading parts of program code to a specialized hardware (neural processing unit) that can efficiently execute those parts. Since neural networks are inherently approximate, they can save considerable amount of energy through simplifications of operations.

1.2.2 Approximate arithmetic blocks

Arithmetic components such as adders and multipliers are the most common targets for approximations in any application. There has been a plenty of works that implement approximate adders and multipliers. One of the most common techniques to generate approximate adders is reducing the hardware complexity of the carry chain and predict the carry using k less significant bits (where k is less than the output width n). Such adders are known as carry speculative adders [35]. Other works suggest segmented adders that are constructed from small adder units working in parallel thereby truncating the carry chain [36, 37, 38].

Approximate multipliers have been mainly developed through three main approaches. The first approach works by approximating partial products such as the one proposed by [26] where a 2×2 multiplier computes the partial product and then used as a block in a bigger multiplier unit. Second approach targets partial product tree in an approximate Wallace-tree multiplier augmented with a carry prediction unit [39]. Third category of works employs approximate adder and compressor units to compute the partial products [40, 41].

A number of publicly available libraries consisting of approximate adders and multipliers have been introduced [38, 42, 43, 44, 45]. The application of these modules in a functional approximation flow is, however, not straight forward since many of these libraries have been constructed for specific target architectures and are not generic. Only a limited number of them can be utilized as off-the-shelf libraries without having to significantly adapt their underlying implementation techniques. An example is the *EvoApproxLib* [46] which is a public library of approximate adders and multipliers. It provides various implementations (such as *C/C++, Verilog, Matlab*) of adders and multipliers of bit-widths 8,12, and 16 (as of writing this thesis) generated with Cartesian genetic programming technique. Moreover the components are characterized with error, area, and power consumption information that can be leveraged to constitute heuristics for the *Design Space Exploration* (DSE).

1.3 The focus of this thesis

This thesis focuses on automated approximate accelerator synthesis at the micro-architecture- / RTL level. The rationale behind choosing this level of abstraction is that it offers a wide range of approximation possibilities since the candidates for approximation i.e., the arithmetic blocks can be identified at this level and the approximation process can also leverage many of the available approximate component libraries.

Another advantage of this level of abstraction is the relationship of error evaluation and the application-level quality information. In most cases the quality of the implementation can be verified on the application-level e.g., the output generated by an image sharpening algorithm implemented at RTL-level. On the other

hand, the approximation techniques applied at the lower level of abstraction i.e., components- / logic-level provide less information about the effect of approximations on the application-level quality. Moreover, the improvements in the target metric at the components- / logic-level are marginal since the approximations affect the smaller cells in the circuit e.g., the logic gates and the nets.

Besides the advantages offered at this level of abstraction, efficient DSE is still a challenge since the availability of large number of possible approximations even for a moderate number of operators in an accelerator circuit can lead to an explosion of combinations. Moreover, in presence of multiple objectives for optimization, the process becomes more complex owing to the fact that some of the objectives are not correlated and are hard to predict with simple heuristics.

1.4 Contributions of the thesis

The contributions made by this thesis are outlined in the following:

- It proposes and implements a novel synthesis algorithm for approximate accelerator synthesis based on *Monte Carlo Tree Search* (MCTS), an intelligent stochastic search algorithm that has been proven successful to problems with large branching factors such as games. Inspired from the outstanding success of MCTS in games, we adopted MCTS for approximate hardware synthesis. However due to various dissimilarities between the two domains, significant transformations had to be done to fully adapt MCTS for approximate accelerator synthesis. To the best of our knowledge, this is the first work that adapts MCTS for approximate hardware accelerator synthesis.

- It proposes and implements two pruning schemes to efficiently explore the design space for approximate accelerator synthesis problem. The pruning schemes are used to recognize parts of search space that lead to designs with larger errors exceeding the allowed error bounds and truncate those paths early in the DSE process.

- It proposes and implements a novel hybrid methodology for approximate accelerator synthesis. The proposed methodology combines analytical approximation techniques and parallel MCTS forest. Through an analytical approximation phase, it prunes a large portion of irrelevant search space without performing costly quality validations and in the following phase, performs a parallel search in different parts of the search space thus providing an efficient DSE methodology.

- It proposes and implements a faster DSE approach based on deep neural networks. Through trained learning models, it avoids a large number of time-costly simulations and achieves similar quality of results in extremely short runtimes as compared to the simulation-based DSE.

1.5 Organization of the remainder

The remainder of this thesis is organized as follows:

Chapter 2 explains underlying concepts of approximate accelerator synthesis and discusses state-of-the-art in the domain of automated approximate accelerator synthesis.

In **Chapter 3**, an automated framework is proposed that leverages MCTS for synthesis of approximate accelerators. Proposed framework is interfaced together with commercial synthesis and simulation tools to provide a complete flow. Later in the chapter, the implementation of MCTS is evaluated on an open-source framework on a number of practical benchmarks with a wider set of parameters.

Chapter 4 proposes a hybrid methodology that combines analytical approximation method with a parallel stochastic search-based optimization (based on MCTS) to perform efficient DSE for approximate accelerator synthesis. The proposed two phase methodology first samples different parts of the search space using analytical bit-width estimation phase and then initiates an MCTS search forest where the designs found in the first phase act as root nodes for the search trees. Phase 2 then performs a parallel DSE from these pre-sampled points thereby enabling more efficient exploration with considerable reduction in the runtime. Results show that the hybrid methodology outperforms purely analytical or purely search-based methodology by achieving higher area savings.

Chapter 5 presents a faster approach to estimate the error of the intermediate approximate designs generated during the DSE phase. The approach aims to speedup the MCTS-based DSE by utilizing fast and accurate deep learning error estimation models. The proposed approach is capable to explore similar number of nodes in the design space in extremely shorter runtime as compared to a simulation-based flow. The quality of results achieved by the proposed approach are equal or even better in some cases.

Chapter 6 concludes the thesis.

Further details and the preliminary concepts of automated approximate accelerator synthesis are explained in Section 2.1 and a detailed comparison and discussion on related works follows in Section 2.2.

Chapter 2

Background and Related Work

This chapter discusses preliminary concepts of the approximate accelerator synthesis and provides an overview of the related work. In the beginning of this chapter, a simplified view of the automated approximate accelerator synthesis flow is presented. Later, key steps required for the *Design Space Exploration* (DSE) of approximate accelerator are discussed. Finally, a comprehensive review of related works concludes the chapter.

2.1 Automated synthesis of approximate accelerators

Given an exact accelerator circuit description (e.g., a circuit described in any HDL language or SystemC) and an error constraint set by the designer, obtaining an approximate accelerator can be described as an optimization problem where the objective is to find design(s) offering better performance parameters e.g., power, delay, and / or area subject to error constraint (E_{th}) i.e., the error at the output must be kept less than E_{th}.

$$\min \ Fit(A) \quad \text{subject to } E(A) < E_{th} \tag{2.1}$$

where $Fit(A)$ is a function that represents the fitness of the approximate design A in terms of its circuit parameters such as area, power and delay and $E(A)$ represents the error of approximate design (computed using any of the error metric defined in Section 2.1.3).

While some of the existing works formulate the approximation and the error propagation through analytical models [47, 48, 49, 50], the majority of the works achieve the approximation through a search-based optimization [15, 16, 51, 14, 19, 17, 52, 21, 23, 22]. The former being centered towards particular approximation method and error metric, offers less flexibility but generally have shorter runtime. The latter is generally applicable to larger set of accelerator circuit types, is more flexible and offers much larger set of possible approximations but is more time-consuming due to large size of solution space.

In a search-based optimization, the original design is incrementally approximated via iterative refinement process. In each iteration of the process, the design

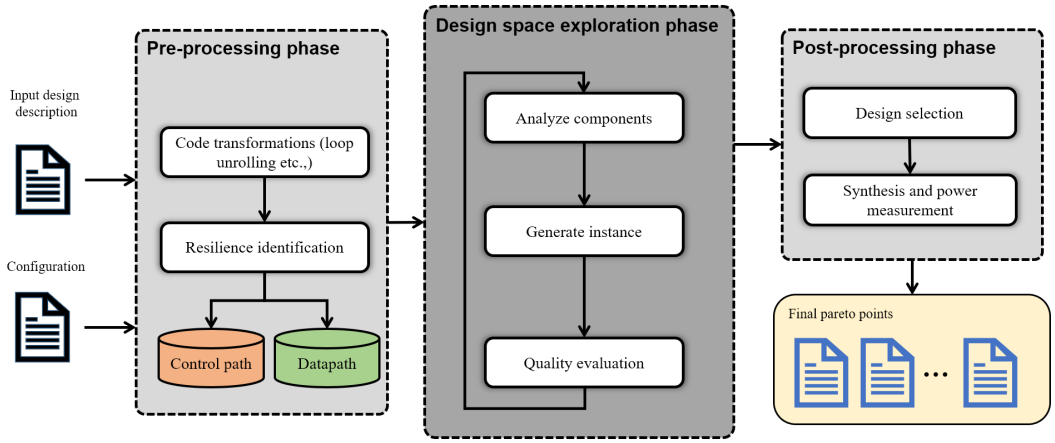


Figure 2.1: Simplified view of the search-based approximate accelerator synthesis flow.

is further approximated with one of the available approximation methods/transformations until either the design could not be further approximated without violating error bound, or the stopping criteria is met. Figure 2.1 shows a simplified view of such flow. The flow starts with the input design (original accelerator circuit description) and a configuration file (optional and can be used to pass values for different parameters to later steps of the flow) which are provided to a preprocessing phase. This phase performs steps such as applying common code transformation e.g., loop unrolling to uncover approximation possibilities and a resilience identification step (among others) to identify the operators suitable to approximation. Examples of resilient code parts are the data-path elements such as adders and multipliers. Parts of code except the data-path elements are marked as control parts and are not considered for approximation since manipulating them might result in undefined behavior of the application. Common methods of resilience identifications are manual code annotations [53] and variables dynamic range computations [54] among others.

The next phase is the DSE and is an iterative process that: (1) identifies a candidate to approximate, (2) generates an approximate design instance by performing approximation according the to the selected candidate (3) evaluates the quality of the solution. The iterative process is typically repeated till the allocated time budget or number of iterations are exhausted. The result is one or more approximate designs that offer better quality parameters and are then forwarded to a post-processing phase which applies further filtration based on some criteria and performs finalization steps. The DSE phase is a crucial block of the flow since its steps such as approximation, quality assurance and search space exploration have a strong impact on the quality of the results. In the following subsections, these steps are explained in more detail.

2.1.1 Approximation instance generation

Generating an approximate instance refers to the process which results in an approximate accelerator variant where one or more candidates have been replaced with approximate components. Very often these approximate components are available to the synthesis framework as a pre-characterized library furnished with the error, area, delay and power consumption information for all components. Such library components can be obtained with a variety of approximation techniques. In fact, state-of-the-art approximate arithmetic components include ones obtained with rather simpler techniques such as truncating the lower bits of adders and multipliers [55], cutting the carry chain of the adders [37], replacing lower parts of the adders with logic *or* [25], etc. Some publicly available component libraries such as EvoApproxLib [46] provide even larger number of implementations of arithmetic units with varying trade-offs among error and performance parameters.

Some frameworks such as [56] generate the approximate components library on-the-go by transforming the hardware description of the selected candidate to a suitable representation such as *And-Inverter Graphs* (AIG) or *Binary Decision Diagrams* (BDDs) and then applying approximation on the selected transformed representation. The approximated component is then transformed back to the original representation and plugged in to the circuit to produce an approximate variant.

With the increased number and a wide range of available approximate components, the task of selecting suitable arithmetic components from the library for substitution during the DSE becomes non-trivial. Moreover in presence of error masking effects, finding best suited combination of approximate components to optimize for target metrics (area and /or power) becomes even more challenging task.

2.1.2 Search space exploration

The search-based DSE process relies on massive search to find and validate possible solutions. The total number of possible solutions i.e., search space is however, very huge and expands exponentially even for medium-sized accelerator circuits as the number of candidates and approximate transformations increase. Since majority of the existing frameworks have represented the search space with a tree where nodes represent different approximate variants generated during the DSE, we will use here an example of a tree to explain how big the search space could get. Let us suppose that the total number of combinations (or approximate variants) that can be realized for an accelerator circuit (and thus the size of the search space) be N, the number of candidates be C, and total approximations that can be applied to a candidate be represented as A, then N would grow as $A^0 + A^1 + A^2 + ... + A^C$ as the tree starts growing in depth, thus resulting in a geometric progression [15]. Mathematically, total number of combinations or N can be represented as in the following equation.

$$N = \frac{1 - A^{C+1}}{1 - A} \tag{2.2}$$

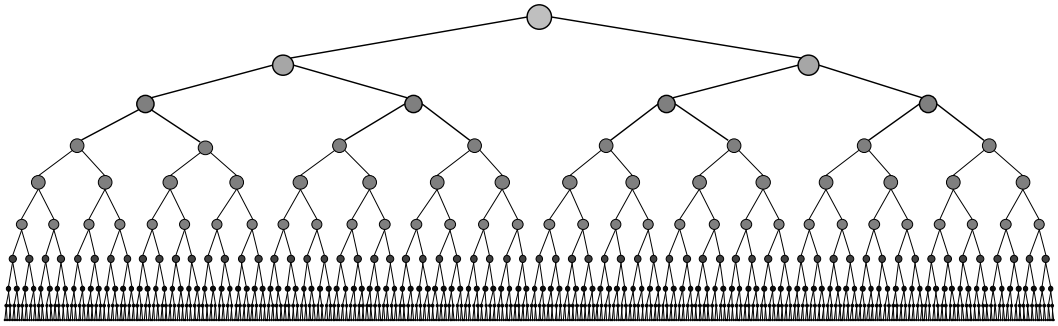


Figure 2.2: Growth of search tree.

To further elaborate on this in a graphical manner, consider an example accelerator circuit that has nine candidates for approximations and two possible transformations available (typically the number of approximate transformations are much higher). Figure 2.2 shows all possible combinations that can be obtained for the accelerator circuit under consideration represented in the form of a search tree. Each node in the tree is a possible configuration or approximate variant of the original accelerator circuit where some (or all) of the candidates are approximated. Note that the search tree grows exponentially in terms of number of nodes as we go down in the depth. Many of the nodes (like the ones shown as leaf nodes) will more likely be not good in terms of output quality as their error will exceed the desired error bound.

With the number of candidates (and the transformations) increasing, evaluating all possible combinations/nodes in the search tree becomes impractical due to the exponential growth in the size of tree. Using the formulation from Equation 2.2, in Figure 2.3 the total number of combinations (or nodes in a tree) against increasing number of candidates (with only three possible transformations for each candidate) is shown. It can be seen that the as the number of candidates increases beyond 14, the total number of nodes rapidly jumps to as high as 21.52×10^6. In fact, considering that the average time taken for a cycle-accurate simulation that applies around 10^6 test vectors to a medium-sized accelerator circuit under test takes around 5 seconds on a regular computer system [57], simulating 21.52×10^6 nodes will take around 3.14 years of computations.

For a commonly used application from signal or image processing domain, the number of candidates and approximate transformations are typically higher than those in the above-mentioned example. Since exploring such a huge search space exhaustively is not practical in a reasonable time, state-of-the-art methods mostly rely on heuristic-based search to explore the design space.

2.1.3 Quality evaluation

The quality evaluation is the key step that validates the approximate designs. Based on the quality evaluation step the DSE can decide in which direction to expand the

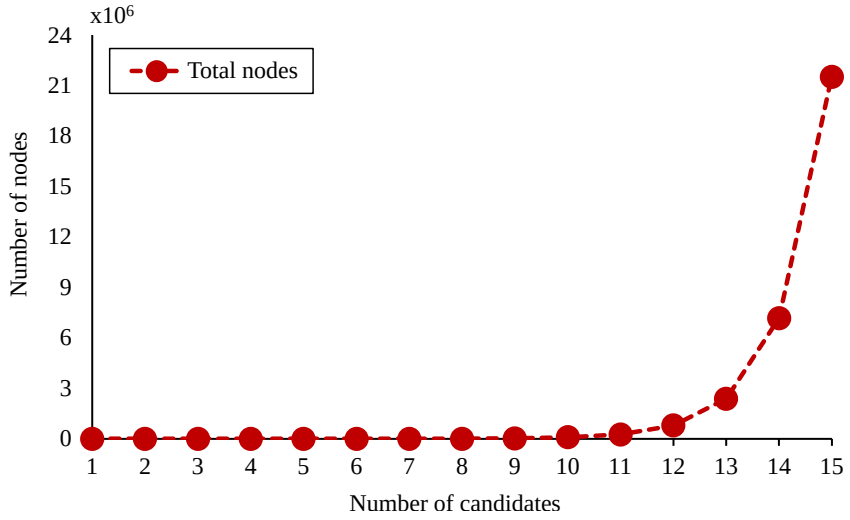


Figure 2.3: Growth of search space.

search space. The exact method to determine the quality of the approximate design depends on the requirements of designer, the selection of the error metric and whether or not a formal proof is required for the quality of the design instance. Common method for quality evaluation for approximate accelerators are *formal verification* and *testing*.

Formal verification refers to methods that provide a proof of equivalence between the specification and implementation of the accelerator circuit. For approximate computing context, this means that the property is relaxed to equivalence to some bound [58]. Although formal verification-based approach can provide strong statement about the quality of the circuit, it also tends to have longer runtime. A common example for formal verification involves an approximation miter circuit that is connected with the outputs of both original and approximate accelerator circuits [17, 59]. The miter compares the outputs and raises the error flag when the output of the approximate accelerator circuit diverges from the output of the original implementation beyond the allowed threshold. The worst-case runtime for formal verification is substantially longer than the testing-based approach especially for larger circuits since the input combinations must be exhaustively evaluated to provide a guarantee over the worst-case error. In addition to that, certain error metrics such as the average-case error are still hard to be verified using the formal verification-based approach [60, 59].

On the other hand, a testing-based approach have shorter runtime but does not provide any proof about the error. This approach works by sampling a reasonably sized subset of input combinations selected with a known statistical distribution and then simulating the approximate accelerator circuit with this input sample. Typical sample sizes used in related works range from 10^4 to 10^6 test vectors [23, 61, 15, 59]. As the test sample is applied to the approximate circuit, the output is sampled and

is then compared with the output of the non-approximated circuit (often referred to as golden result). Typical average case error metrics such as mean relative error or mean absolute error can then be evaluated from the comparison to represent the amount of error rather than providing an upper-bound on the output error.

Despite being faster, the accuracy of the testing-based approach can vary depending on factors such as input data distribution or the size of the test vectors. It should be noted that the total number of input combinations from which the test vectors are to be sampled are 2^n where n represents the width of the primary inputs. As an example, consider an adder that performs addition of two 32-bit numbers. Then the total number of input combinations would be 2^{64} (where each input can range from 0 to $2^{32} - 1$).

In the following, some of the common error metrics used in regard of the approximate accelerator synthesis are defined.

Error metrics

Approximate solutions generated by any of the techniques mentioned in the previous section need to be characterized in terms of their quality in order to be ranked in an approximate accelerator synthesis flow. This is done by evaluating the error generated by them. Different error metrics are used to represent the amount of approximation error. Generally, the error metrics can be classified as either worst-case or average-case. The former type is more common in methods that employ formal verification to evaluate approximate designs. The latter type is obtained typically through applying test vectors (e.g., 10^6 randomly generated input combinations) to the design under test to estimate the value of error. Average-case error metrics are more often preferred since they are suitable for many application domains of approximate computing e.g., image processing where the average-case can provide more insight about the overall quality of the output rather than just providing the worst-case information. In the following, some of the commonly used error metrics are defined.

Notations: *Consider a Boolean function F with n as the width of primary input and m as the width of primary output. F is then defined as a mapping of input range* $\mathbb{B}^n$ *to output range* $\mathbb{B}^m$ *where* $\mathbb{B} = \{0, 1\}$. *The approximate version of F is defined similarly and denoted as* $\hat{F}$. *Let* $O(x)$ *be the output of F and* $\hat{O}(x)$ *be the output of* $\hat{F}$ *on a given input x.*

We can then define the following error metrics.

1. *Worst case absolute error (E_{wce}):* The worst-case absolute error is computed as the maximum difference between original and approximate outputs for all input combinations and is defined as the following:

$$E_{wce} = \max_{\forall x \in \mathbb{B}^n} |O(x) - \hat{O}(x)| \tag{2.3}$$

2. *Worst case relative error (E_{wcre}):* The worst-case relative error gives the maximum relative difference between original and approximate outputs for all input combinations and is defined as the following:

$$E_{wcre} = \max_{\forall x \in \mathbb{B}^n} \frac{|O(x) - \hat{O}(x)|}{\max(1, O(x))} \tag{2.4}$$

Note that $\max(1, O(x))$ *is used to avoid divide by zero when* $O(x) = 0$.

3. *Bit-flip error (E_{bf}):* The bit-flip error is defined as the maximum number of different bits in the output for all input combinations. It is sometimes known as the worst-case *Hamming distance*. It is defined as the following:

$$E_{bf} = \max_{\forall x \in \mathbb{B}^n} \left(\sum_{i=0}^{m-1} O_i(x) \oplus \hat{O}_i(x) \right) \tag{2.5}$$

4. *Mean absolute error (E_{mae}):* The mean absolute error is the average of the sum of absolute differences between original and approximate output for all input combinations and is defined as the following:.

$$E_{mae} = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} |O(x) - \hat{O}(x)| \tag{2.6}$$

5. *Mean relative error (E_{mre}):* The mean relative error is the average of the sum of relative differences between original and approximate output for all input combinations and is defined as the following:

$$E_{mre} = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} \frac{|O(x) - \hat{O}(x)|}{\max(1, O(x))} \tag{2.7}$$

Note that $\max(1, O(x))$ *is used to avoid divide by zero when* $O(x) = 0$.

6. *Mean relative error percentage ($E_{mre(\%)}$):* The mean relative error percentage is the normalized E_{mre} over primary output m and expressed as a percentage. It is defined as following:

$$E_{mre(\%)} = \frac{1}{2^m} E_{mre} \times 100 \tag{2.8}$$

7. *Error rate (E_{er}):* The error rate or error probability is the number of approximate outputs that are different from original output averaged over all input combinations. It is defined as the following:

$$E_{er} = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} X \tag{2.9}$$

where,

$$X = \begin{Bmatrix} 1 & if\ O(x) \neq \hat{O}(x) \\ 0 & otherwise \end{Bmatrix}$$

8. *Mean squared error (E_{mse}):* The mean squared error is the average of the sum of the squared differences between original and approximate outputs and is defined as the following:

$$E_{mse} = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} (O(x) - \hat{O}(x))^2 \tag{2.10}$$

9. *Peak signal to noise ratio (E_{psnr}):* The peak signal to noise ratio is obtained from MSE and is very often used in image processing applications and defined as:

$$E_{psnr} = 20 \cdot \log(10) \left(\frac{MAX^2}{MSE} \right) \tag{2.11}$$

where MAX refers to the maximum value of the output.

The next section provides a review of related works that deals with the approximate accelerator synthesis problem using different search-based approaches. In the end of the section, other analytical techniques that do not require search, are also discussed.

2.2 Related works

This section provides a review of the state-of-the-art in the synthesis of approximate accelerators. We divide the works in this domain in two main categories i.e., search-based methods and analytical methods. The former uses arithmetic component substitution in an iterative refinement process whereas the later focuses on analytical formulation to determine the approximation parameters without performing iterative search.

2.2.1 Search-based methods

The shared characteristic of previous works is the use of various heuristic-based search techniques to either explore the best possible approximate transformations needed for the target accelerator circuit or examine various approximate transformations via Boolean optimization while maintaining a quality constraint. These include greedy, branch-and-bound, binary search, and genetic-based algorithms.

The ABACUS framework [15] is an example that uses a greedy approach to explore the design space to generate approximate accelerators. It picks one approximate transformation that appears to be the best in the current iteration and expands the search space from that point. Although it has shorter runtime it is most likely to

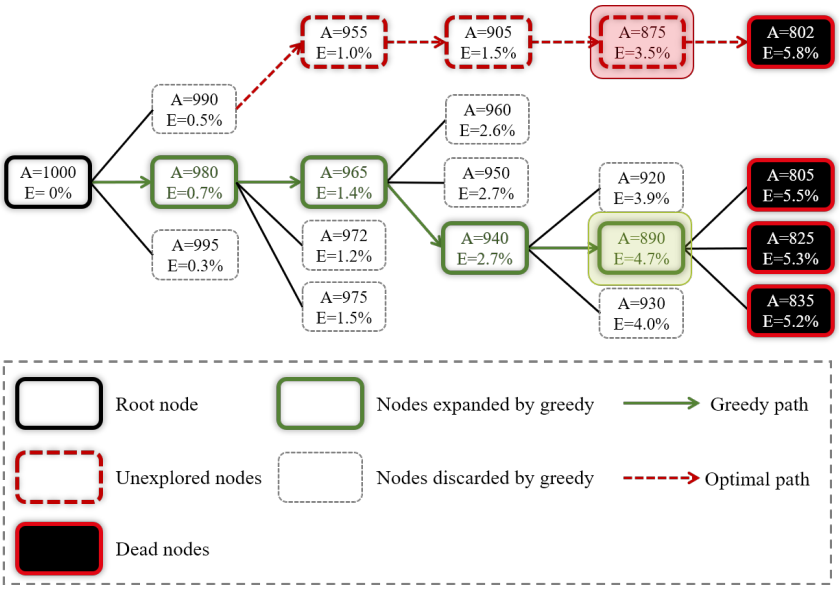


Figure 2.4: Greedy-based search [15].

miss the global optimum as it strictly confines the search space in early steps of the algorithm and therefore the framework becomes remarkably prone to overlook the global optimum. Moreover, the quality of the final results heavily depends on the moves taken in the first few iterations as there is no possibility for a backtrack.

This can be explained with an example in Figure 2.4 where a greedy-based search space exploration flow is shown. Here, each node represents an accelerator circuit configuration and the area and error information are shown inside each node. The greedy process starts with the root node (original circuit) and in each iteration expands the current node by applying a number of approximate transformations on a selected candidate. It then picks the node that has best area saving and expand this node in the next iteration. The greedy process stops when there are no more nodes having error less than 5% (target error bound for this example) among the children of current node and reports the current node as the best possible approximate accelerator circuit found. In this example, the path followed by the greedy approach is shown by the green solid arrows and leads to an approximate accelerator circuit with an area value of 890 units and 4.7% error. However, the optimal path in this case is the path shown by the dotted red arrows that could have led to a design having 875 units of area and 3.5% error.

The extended version of ABACUS [16] uses a mix of greedy and genetic algorithm (NSGA-II). In each iteration, it generates a population of actions and then selects multiple nodes to expand the search space. This, of course, improves the quality at the cost of higher computational complexity yet the iterative nature of ABACUS may cause a set of approximations, belonging to the former generations, dominate the global search space and hence limit the selection of other promising approximations that might appear in later generations.

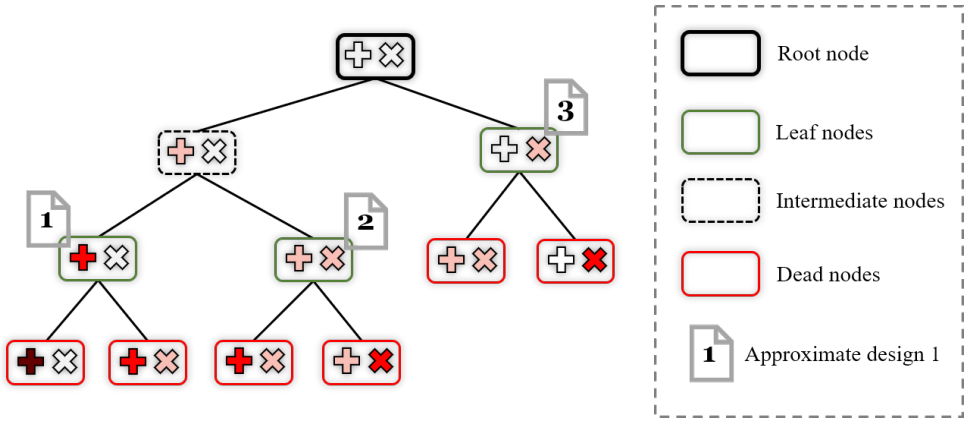


Figure 2.5: Branch-and-bound (B&B) search [51].

Barbareschi et al., in [51] implement their synthesis approach with a branch-and-bound based DSE tool called IDEA. Their algorithm expands search space for one candidate as deep as possible and then backtracks. In this manner, IDEA iterates over the possible approximations of each node in the design and performs a depth-first search until no more approximations are possible and then backtracks. The performance of this technique to generate an optimal approximate design is limited since the large portion of the search budget is dedicated to identifying the best possible approximation of the firstly visited nodes of the design.

Figure 2.5 shows graphically how such a branch-and-bound based approach proceeds with iterative approximation for two candidates i.e., an adder and a multiplier. The left branch of each node represents the approximation path of the adder and right branch shows approximation path of the multiplier. The approximate configuration of each node is inside the node. The darker color represents more intensive approximation level for an operator. This is with regard to the IDEA where the approximations are applied in the form of precision scaling hence deeper nodes generally have less precision in one or more of their operations. The backtrack occurs when the approximation results in the violation of the error bound (shown as red nodes in Figure 2.5). The algorithm continues to build approximation paths as long as the computational budget is available. However, when the number of candidates and possible levels of approximation are considerably larger, a branch-and-bound approach will spend most of the time in just first few branches and ultimately miss a large portion of search space.

Chandrasekharan et al. [14] propose approximation technique for Boolean functions represented as *AND Inverter Graphs* (AIGs). The technique called AIG rewriting first identifies the critical paths of the circuit and then iteratively selects and replaces the cuts on the critical paths using a greedy approach. The cut selection scheme assumes that starting approximation transformations with a smaller size cut can result in better approximations hence it sorts the cuts in an increasing order before selection. The error constraint is then formally verified through a SAT solver. With

a similar objective, circuit carving [19] aims to prune the maximum part of an exact design represented as a *Direct Acyclic Graph* (DAG) by finding a cut that has the maximum number of gates. The cut formation relies on a weight labeling step that follows binary search algorithm to decide whether or not to include a DAG node in the cut. The approximate design is then obtained by removing that cut from the design. Circuit carving faces the combinatorial explosion in case of large size circuits not only for the search but also for the weight labeling of the nodes in the DAG.

The ASLAN framework [17] by Ranjan et al., presents yet another approach for automated synthesis of sequential circuits. ASLAN introduces a so-called *quality function* that encodes quality constraint to measure the precision loss of the multi-cycle circuits via formal verification. To identify the type of approximation, that leads to the highest power and area efficiency, a tremendous amount of exploration over the large space of possible transformations is required. ASLAN uses a gradient-based search technique which needs a large simulation budget for bigger designs, and may get stuck in the local optimums for highly non-linear quality constraints of circuits.

In another work, a substitute-and-simplify method (SASIMI) [21] is proposed that achieves approximation by determining pairs of circuit nodes that have similar functionalities and then substituting one with other. As a result, the logic required for one of the signals can be eliminated. SASIMI uses a heuristic ranking method based on the area and delay of the signals. The circuit is then constantly simplified using the ranking heuristics until the error bound is reached.

In a more recent work [23] SCALS framework iteratively exploits three simple transformations (e.g., adding extra logic gates among the circuit nodes, randomly changing the type of logic gates, and flipping signal polarities) on the sub-networks extracted from the original circuit. The candidate nodes for approximation are selected randomly, and *Markov Chain Monte Carlo* (MCMC) sampling is used to accept or reject the recently applied transformation.

BLASYS framework [22] by Hashemi et al., uses Boolean matrix factorization technique to decompose the truth table entries for extracted logic parts of the circuit to simpler approximate versions and then substitutes them for original parts to yield approximate instances. The DSE phase relies on a greedy search to find suitable combinations of the decomposed matrices that provide better error-area trade-offs.

The *Jump Search* [18] proposes a greedy-like approach to quickly select a path in the design space based on area and error heuristics followed by a validation step to find the top quality verified node on the path. The proposed approach is faster than many other similar greedy-based approaches because it does not invoke synthesis during the path selection rather relies on a composite heuristics function that combines the error and area impact of the selected candidates on the path. For the very same reason, the performance improvements can sometimes be suboptimal due to poor selection sequence of the candidates.

Table 2.1: Categorization of search-based frameworks

Framework	DSE method	Approximation method	Open-source?
ABACUS [15]	Greedy, Greedy+NSGA-II	AST transformations	Yes
AIG rewriting [14]	Greedy	AIG rewriting	Yes
ASLAN [17]	Greedy	Precision scaling	No
BLASYS [22]	Greedy	Boolean matrix factorization	Yes
Jump Search [18]	Greedy	Precision scaling	Yes
CC [19]	Binary Search	Logic removal	No
IDEA [51]	Branch & Bound	Precision scaling	No
SCALS [23]	Metropolis Hasting	Logic transformations	No
SASIMI [21]	Hill-climbing	Substitute and simplify	No

Table 2.1 lists the DSE methods and approximation techniques used by the mentioned frameworks. Only four of the mentioned nine frameworks have been made open source.

2.2.2 Analytical methods

The second category of works relies on either analytical formulation of the quality metric or similar techniques to generate approximate accelerators. This category of techniques does not require massive search and is often faster than the search-based methods. Nonetheless the applicability of such methods is limited since they are not much flexible and scalable. In the following, a brief review of state-of-the-art analytical techniques is provided.

A number of methods in this category exploit statistical properties to model the effect of error on the overall application quality. For instance, Huang et al., in [47] use a modified version of interval arithmetic to propagate the intervals of variables up to the primary outputs. The non-linear behavior of the error propagation is taken into account by considering the correlation among the variables through affine arithmetic. This type of error propagation however results in a pessimistic estimation of the output error.

Another analytical framework is presented in [48] that computes PSNR values of an image processing applications composed of specific inexact adder types. Three different adder types were considered, and their error distance (ED) values were modeled using the proposed framework.

In [49], authors provide a methodology to characterize variance of the output error in a direct acyclic graph (DAG). The DAG is composed of nodes representing the imprecise adders having different number of their LSB's approximated. Then the output variance is computed from the number of approximated LSBs and their respective error distributions that were computed with exhaustive simulations.

Sengupta et al. in [50] propose an approach that is more generic and applicable to larger set of imprecise operators. The approach uses Mellin and Fourier transforms

to compute the probability mass function (PMF) of the output error for operators in a DAG representation. For larger benchmarks, the applicability of the approach is however questionable due to the exponential growth of the computational complexity.

Mrazek et al. [62] employed machine learning techniques in their framework *autoAx* to obtain the configurations for the bit-widths of a set of adders and multipliers to be used in an image processing application.

Venkataramani et al. in [52] apply Boolean don't care optimization via gradient descent for the approximation of combinational circuits. The proposed framework SALSA transforms the approximate logic synthesis to a traditional logic synthesis problem that can use the existing synthesis tools to generate "correct by construction" approximate circuits. A user-defined quality constraint circuit is to be constructed for any given circuit to be synthesized. This demands for domain-specific expertise since the so-called *Q-function* required to have quality constraints encoded inside it. The quality constraints are then evaluated exhaustively to recognize the set of don't care inputs which can then be simplified via the traditional synthesis tools.

Chapter 3

MCTS-based Framework for Approximate Accelerator Synthesis

3.1 Chapter overview

This chapter is organized in two main parts. In the first half, a comprehensive framework for automated synthesis of *Approximate Accelerator Circuits* (AxACs) from either structural or behavioral descriptions is proposed. The framework adapts *Monte Carlo Tree Search* (MCTS), as a stochastic search technique, to deal with the large *Design Space Exploration* (DSE), which enables a broader range of potential possible approximations through lightweight random simulations. The proposed framework is able to approximate the design Pareto set even with low computational budgets. Experimental results highlight the capabilities of the proposed synthesis framework by resulting in up to 61.69% energy saving than the exact circuit while maintaining the predefined quality constraints.

Later in the second half of the chapter, a joint work with other colleagues from our group on the development of CIRCA (an open-source AxAC synthesis framework) is presented. The motivation behind CIRCA is to provide a framework on which new approximation, search, and validation techniques can be rapidly developed and evaluated. To demonstrate the modularity and generality of CIRCA, and also to show how MCTS algorithm can be plugged in to any generic AxAC synthesis framework, we implemented MCTS as a search method in CIRCA, coupled it with a formal verification flow, and performed experiments to compare multiple search techniques and two different approximation methods (precision scaling and And-Inverter-Graph rewriting) on a larger set of benchmarks. Experimental results with CIRCA reveal important insights about the role of search and approximation methods on the quality of the AxACs generated.

The results of CIRCA framework in general show that even with a very small computational budget (100 iterations), MCTS was able to achieve better performance than a greedy-based method, and close to the simulated annealing search method in most cases. However, with too small computational budget, MCTS was not able to collect sufficient statistics to train its selection policy and thus its performance resembled to that of random sampling. It should be noted that the experimental setup

of CIRCA was substantially different than the MCTS-based framework proposed in the first half of this chapter and so were the competing search methods in both cases. Further, we believe that selecting a reasonable computational budget has strong impact on MCTS' performance. To this end, we provide some further discussion at the end of Section 3.3 to discuss the dissimilarities between the two setups and how the choice of a search budget and other parameters can affect the performance of MCTS.

In the following, the main contributions of this chapter are listed:

- An MCTS-based framework is proposed that presents a new success for efficiently handling the problem of DSE for automated AxAC synthesis even with a tight computational budget.
- The underlying search technique is modified to prune all the combinations that are not promising for the target design in terms of quality. Using this, it avoids the combinatorial explosion of inadequate possible approximations.
- The benefit of using the proposed MCTS-based framework is validated in circuit energy and area for five widely used benchmark accelerator circuits in various embedded applications. We also compare the obtained results with that of other AxAC synthesis tools to illustrate the effectiveness of the proposed framework.
- The proposed search technique is implemented in a modular and flexible open-source framework CIRCA and is evaluated with two other search techniques with two different approximation methods on a larger set of benchmarks.

The remainder of this chapter is organized in three sections. First, Section 3.2 provides the details of the proposed MCTS-based framework and experimental results. Second, we provide details of the open-source framework CIRCA along-with comprehensive results of MCTS with other search techniques in Section 3.3. At the end of this section, we provide a detailed discussion about the impact of search budget on the performance improvement offered by MCTS. At last, in Section 3.4 we conclude the chapter and discuss future directions.

3.2 Proposed MCTS-based framework for approximate accelerator synthesis

This section explains the proposed framework for synthesis of AxACs. First, we describe the work-flow of the proposed framework. This is followed by a brief review of the MCTS algorithm and the explanation of its modifications to be used in the proposed framework.

3.2.1 Overview of the framework

The overall flow of the MCTS-based AxAC synthesis framework is illustrated in Figure 3.1. The framework is fed with the behavioral or structural description of the exact circuit. The framework exploits an HDL parser [63] in order to identify the target operations and operands for approximation (the preprocessing step in Figure 3.1). As a result, the obtained parsing tree is later modified by the search engine to generate a set of AxACs. In our framework, we use the approximate transformations such as precision reduction for arithmetic operations and memory elements, broken carry chain (e.g., ACA [37]) for additions and multiplications, operator relaxation (e.g., partially or completely replacing additions with *OR* or *XOR* operations) and loop unrolling (which uncovers the possibilities for further approximations for behavioral descriptions).

The DSE phase of the framework starts with extracted components being configured as the root node of the search tree by the MCTS engine shown in Figure 3.1. A new approximate design variant is then generated by applying an available approximate transformation that is selected from a component library containing approximate modules. This design variant is evaluated with a batch of test vectors selected randomly from the training dataset. At the end of each test, the framework evaluates the quality of the obtained AxAC through a statistical hypothesis testing using the student's T-test [64]. For a user-defined error boundary such as Er and a confidence level such as Cl, the problem is formulated as the following:

$$\begin{cases} H_0 : & training\,error > Er \\ H_1 : & training\,error < Er \end{cases} \tag{3.1}$$

where H_0 and H_1 are the null and alternative hypotheses, respectively. To be sure that the error of the approximation remains within the boundary, the null hypothesis is evaluated by using a batch from training set. By comparing the probability value for each batch regarding to the Cl value, we can evaluate that the null hypothesis remains true or not. In each iteration, we update the result of the search tree based on the conclusion of hypotheses over each batch. If the design fulfills the quality criteria, it is added as a node in the search tree. The node keeps the circuit configuration of the design variant, the error value and a number of MCTS specific data fields (explained in the next subsection) that are: reward value, number of visits, and pointers to the parent and children nodes. The search engine iteratively continues to build the tree until the computational budget assigned to it is exhausted. The result of the search engine is the set of approximate variants with various circuit characteristics. The set of approximate variants is further evaluated using additional quality metrics to find a set of promising approximate designs (post-processing step of Figure 3.1). This is done to avoid over-fitting of approximate design to the input data. This process is then followed by circuit synthesis, determining area, delay and power consumption. The suitable design candidates are presented as the set of

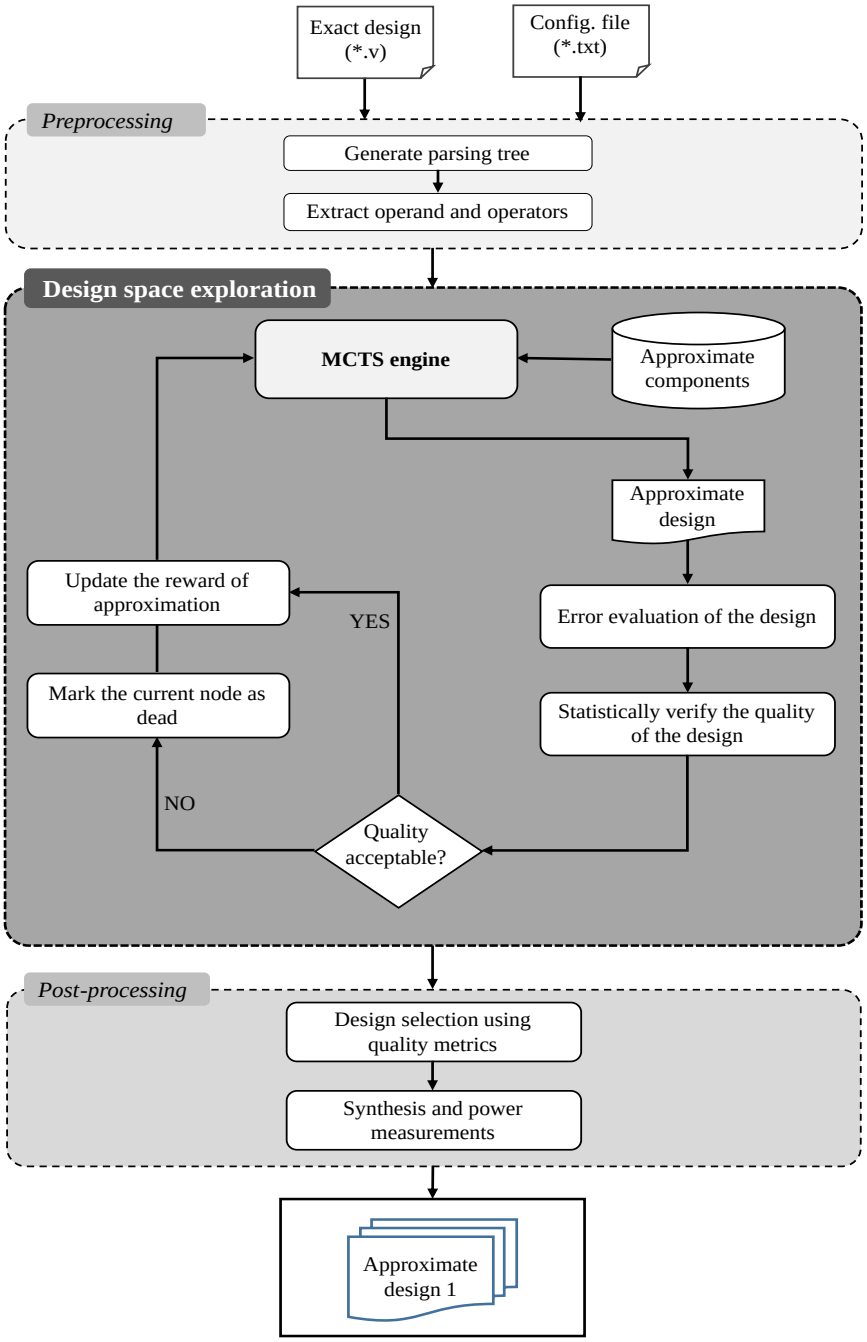


Figure 3.1: Overview of the proposed framework.

approximated Pareto points to the designer.

In the following section we provide the details of the adapted MCTS-based framework and discuss the modifications to improve the performance of MCTS in AxAC synthesis domain.

3.2.2 MCTS and its adaptation to the accelerator circuit synthesis

MCTS is a popular learning-based stochastic search algorithm that uses random lightweight simulations to intelligently build a search tree. In the recent past, MCTS

has received an ample amount of attraction due to its success in the domain of games and a wide variety of other search-based problems such as automated story generation and planning [65, 66]. The ability of MCTS to provide efficient DSE and a balanced tree selection policy makes it an obvious choice for the synthesis problem which has a very large branching factor.

However, the conventional MCTS algorithm [65] needs substantial modifications in order to be applied to the accelerator circuit synthesis problem due to a number of reasons. First, the simulation time of a circuit instance in comparison to the games (e.g., Computer Go) is very large. A play-out in Go for example, completes in a few microseconds where as the cycle-accurate simulation for even a small circuit could take up to a few seconds. Also, unlike a definitive win or loss reward, the reward value for a circuit instance is based on circuit characteristics (area or power) or quality (error). Finally, the design space may contain certain approximation choices that can lead to large errors and therefore, further approximations from those choices onward will not provide any benefit. This is not the case in the games where some initial bad moves can still lead to a win. The modifications done to the conventional MCTS to use it for accelerator circuit synthesis problem are discussed in detail at the end of this section.

The detailed flow of the proposed synthesis framework is represented in Algorithm 1. The algorithm initiates with the description of the original circuit and builds a search tree in which the branches represent different approximation paths and the leaf nodes specify AxAC instances. In each iteration, a circuit instance is selected in the search tree (or added if it is the first iteration). One of the exact operators in this circuit instance is replaced with the approximate operator to produce an approximate version of the circuit. The new AxAC then is added to the search tree (the blue node in Figure 3.2).

The next step is circuit simulation in which a random play-out has to be run. In the game domain, the random play-out means applying actions in random sequences to reach a terminal state (the state which finishes the game). The play-out results in a reward value which in turn is an indication whether the play-out resulted in a win, loss or draw situation. For a circuit instance though, this means the circuit instance undergoes a validation with a random test set and then the reward value is estimated on the basis of its circuit characteristics. The circuit instance is added to the search tree if the error is under the maximum allowed error ε_{max}(the green nodes in Figure 3.2); otherwise it is marked as a dead node (the red nodes in Figure 3.2) in the search tree (Algorithm 1- line 18).

Finally, in the update step the error value (ε) is then updated for the newly added circuit instance in the search tree (Figure 3.2: Update step). This will complete one iteration of MCTS and depending upon the computational budget, the search tree continues to grow with each iteration. The algorithm terminates when the computational budget assigned to it is exhausted.

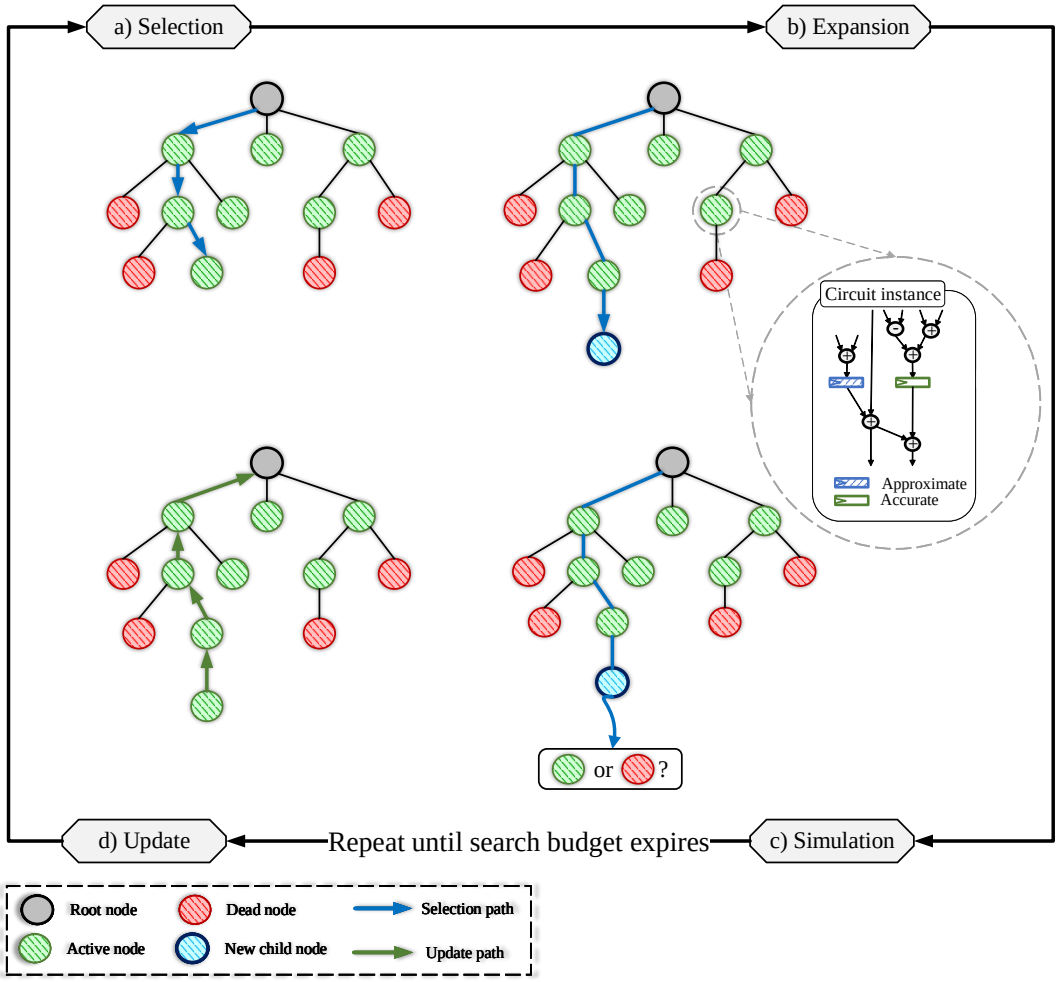


Figure 3.2: Major steps of the MCTS algorithm in the proposed framework.

To have a control on the number of branches in the search tree, we use a decision function (explained in the following subsection) that determines the number of branches to add at each exact operator location by looking at already available branches and the total number of simulations done at that time. This makes sure that the computational budget is fairly utilized by allowing a reasonable number of branches in the search tree (Algorithm 1- line 5). The weighted heuristic for pool components can guide the expansion step to select promising approximate operators (Algorithm 1- line 9). In addition, node pruning is used to cut the branches that violate the error constraint ε (Algorithm 1- line 13).

In the following, we briefly explain the modifications in our MCTS search engine for the synthesis of AxACs.

Selection policy

The selection policy in the MCTS plays an important role in building the search tree. The *Upper Confidence Bound applied to Trees* (UCT) is the most common formula to select the child node in the search tree [67]. It selects the child node i that has the

Algorithm 1: The MCTS-based synthesis framework

Input: O= Verilog description of the accelerator circuit , M= Computation budget ,
ε_{max} = pre-defined quality threshold
Output: S={Approximated Pareto set of approximate accelerator circuits A_i}

```
1  K ← Circuit components;
2  Initialize search tree (root.config ← K);
3  while M > 0 do
4      node ← root;
5      while (f(S, L)==0 OR node!=leaf) do
6          node ← UCTSelect(node.children);
7      end
8      if (node is not fully approximated AND node is not dead) then
9          a ← approximation chosen with weighted random heuristic;
10         add node_new as a child of the node;
11         node_new.config ← update with a;
12     else
13         backpropogate 0 as reward; // Leaf node reached
14         continue;
15     end
16     ε ← Simulate node_new and compute the error (ε);
17     if (ε > ε_max) then
18         mark node_new as a dead node;
19     else
20         calculate the reward R(ε);
21         S_ActiveNodes ← S_ActiveNodes ∪ node_new;
22     end
23     backpropagate the reward R(ε);
24     M ← M − 1;
25 end
26 S ← Approximated Pareto points from S_ActiveNodes;
27 return S
```

maximum UCT value as defined in the following equation:

$$UCT_i = \frac{W_i}{V_i} + C\sqrt{\frac{\ln(V_N)}{V_i}} \tag{3.2}$$

where W_i represents the reward value for the current node, V_i shows the number of visits for the current node, V_N shows the number of visits for the parent node, and C is the exploration constant which takes a small positive value [65].

The UCT formula has an important feature of keeping a balance between selecting nodes that proved promising in the previous iterations and the nodes which have not been explored yet. This property of UCT suits very well to the DSE of accelerator circuit synthesis since it is equally important to explore the new paths in the MCTS to find new approximation possibilities (exploration) as well as to expand

the previously explored paths (exploitation) for further approximations.

Search space expansion policy

Since the size of the search space grows exponentially with increasing number of candidates and transformations (as explained in Section 2.1.2), any heuristic-based search space exploration algorithm including MCTS would not able to explore whole search space even for medium-sized accelerator circuits. Given the testing-based validation times consuming few seconds for cycle-accurate simulations, exploring a few thousand nodes could take up to hours of computational time. This puts a practical limit on the number of nodes/iterations that a search algorithm can explore.

Moreover, the branching factor of the MCTS tree depends on the number of available approximate transformations. For example, for the *FIR filter* benchmark, we used 27 adders, 24 multipliers, and 18 register modules as the approximate transformations. With this large branching factor, it means that MCTS could not get much deeper in the search tree since conventional MCTS algorithm fully expands a node before expanding any of its children node. In AxAC synthesis however, it is more desirable to have nodes that lie deep in the search tree since this would imply that we get an AxAC with more approximations applied thus potentially more area savings. Keeping this in view, we have to find a balance between adding new nodes to the breadth vs. adding new nodes to the depth of the tree. The formulation should also consider the allocated search budget so that as the search proceeds, deeper nodes are preferred for expansion. To this end, we introduce a heuristic decision function $f(S, L)$ to decide whether a new child node should be added to the search tree. The decision function has a Boolean output and is defined as:

$$f(S,L) = \begin{cases} 1 & \text{if } \frac{S^{\alpha}}{(L+1)} > n_c \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

where S represents the total number of simulations done (number of nodes in the tree) so far, L represents the depth of the tree at the current node, $\alpha \in [0, 1]$ is the expansion parameter, and n_c represents the number of children nodes for the current node.

In the above formulation, the parameter α can be used to tune the behavior of the expansion policy of the MCTS. With lower values of alpha, the expansion step would expand nodes after every few simulations passed, whereas with higher value of alpha, expansion happens after larger intervals. The former is suitable for lower computational budget while the latter supports higher computational budget. More details on the impact of the α and the exploration parameter C on the quality of the results is further discussed in Section 3.2.3.

Table 3.1: Benchmark accelerator circuits

Benchmark	I/O	Candidates	Trans.†	Area (μ^2)*	Power (mW)*	QoR metric§
Conv_filter	32/32	29	69	16 192.98	8.73	PSNR($\geq$ 25)
DCT	22/22	100	44	17 088.92	6.62	MSE(<5%)
FIR	32/64	25	69	15 309.82	6.71	MSE(<5%)
FFT	32/32	200	56	94 944.74	19.47	MSE(<5%)
IIR	32/64	45	69	30 093.63	16.83	MSE(<5%)

† Approximate transformations e.g., precision scaling, carry chain reduction.
§ Error metric used for quality of results.
* The area and power of the accelerator circuits were measured using Synopsys Design compiler using a 22nm technology library

Selection of approximate transformations

The framework also utilizes weighted random heuristic to intelligently select the approximate transformations. All the transformations have weights computed using Equation 3.4 which determines their probabilities for the selection during the search process.

$$W = \gamma \times Area + \lambda \times Power\,Consumption \tag{3.4}$$

where γ and λ are the coefficients that can be set according to a designer's preference.

The area and power of the individual approximate components were measured using Synopsys Design Compiler with 22nm industrial technology library where for the power consumption, a uniformly distributed random input workload was applied.

Reward function

In MCTS for games, the random playout results in win, loss or a draw state which is then quantified as a discrete value of 1, -1, or 0. This representation cannot be directly applied to the synthesis problem at hand since the reward of a node in this case describes the precision of its computation. So, we need a function to map the error value to the equivalent reward value. This means that the higher values for measured error (ε) will have small rewards and the lower error values will result in large reward values. In this way, the reward value can be obtained by applying a strictly decreasing function over this interval since we only care about the error value in the interval $[0, \varepsilon_{max}]$. Without loss of generality, we will use quadratic function to calculate rewards (i.e., $R(\varepsilon) = (\varepsilon - \varepsilon_{max})^2$).

3.2.3 Experimental setup and results

Setup of experiments

To evaluate the proposed framework, we applied it to five benchmark accelerator circuits selected from several domains of applications. These benchmarks are briefly explained in the following:

- *Conv_filter* is a configurable convolution filter commonly used for various image processing operations such as image sharpening or image smoothing. In our experiments we used it for image smoothing of 256×256 RGB colored images.
- *DCT* is an eight-stage pipelined *Discrete Cosine Transform* block that has an input of 22-bit fixed-point numbers and performs one-dimensional DCT transformation on the input data. The output is also represented as 22-bit wide fixed point.
- *FIR* implements an eight-tap Gaussian low-pass *Finite Impulse Response* filter, which is commonly used in the domain of signal processing. It takes a 32-bit wide data input and provides a 64-bit wide output.
- *FFT* performs an 8-point *Fast Fourier Transform* operation which is a common processing block for signal and image processing applications. It accepts eight data points represented as 32-bit fixed point numbers and provides eight output 32-bit fixed-point values.
- *IIR* implements an *Infinite Impulse Response* filter that accepts 32-bit wide input data and produces 64-bit wide output data.

Detailed characteristics of each benchmark are shown in Table 3.1. Number of candidates and transformations refer to the total number of exact operators which are to be targeted for the approximation and the number of available approximate operators with different energy/area trade-offs, respectively. It also shows the original area and power consumption of each benchmark. Each accelerator circuit is evaluated by a dataset which is used both during synthesis process for quality evaluation and after synthesis for design validation purposes. The datasets are generated through uniform distribution of random numbers over the whole input range. The quality evaluation is achieved via random sampling, described in Section 3.2, by using a batch size of 2000 samples for training data in each epoch. This will guarantee that the obtained approximate variants do not over-fit to a specific dataset. We certify the validation results through statistical student's T-test with a *Cl* of 95% to provide the statistical guarantee about the quality of generated circuits.

We implemented the proposed MCTS core in C++. The core wrapped by several Python-based modules altogether make a custom tool for the synthesis of AxACs. The modules evaluate the benchmark circuits to identify the candidate nodes for

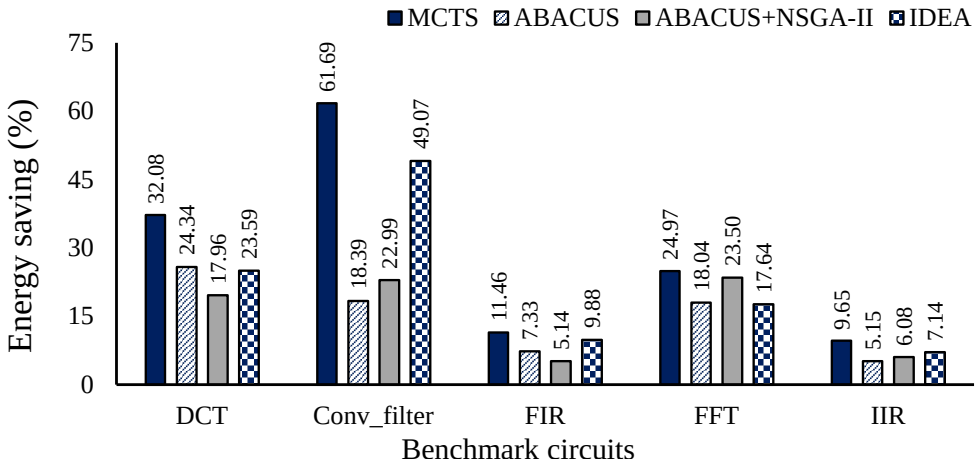


Figure 3.3: Energy savings comparison obtained for various benchmarks using the proposed framework.

approximation, provide approximate instances of the target benchmarks and also connect the core to a standard CAD tool flow which includes *Mentor Graphics QuestaSim* and *Synopsys Design Compiler*. Moreover, a 22 nm industrial technology library was used in our experiments. In our framework, only the computational parts of the data-path are used for approximation, and the control parts are left unchanged. The experiments were run on a computer equipped with two Intel® Xeon® X5650 processors with 47 GB of memory and running CentOS (release 6.7) operating system.

Results and discussion

Figure 3.3 represents the energy savings of the proposed framework over the five benchmark accelerator circuits and compares its results with those of three other competitors (ABACUS, ABACUS+NSGA-II, and IDEA discussed in Section 2.2). All techniques were given the same computational effort in terms of the number of iterations. A total of 10,000 iterations were run for the FIR and IIR designs and 1,000 iterations were run for convolution filter, FFT filter, and DCT block. The average run time per iteration was 14, 15, 518, 203 and 172 seconds for FIR, IIR, convolution filter, FFT, and DCT respectively on a 2.67 GHz Intel® Xeon® X5650 processor with 47 GB of RAM. The values of λ and γ are selected as 0.5 each in all experiments. The hyper parameters C, and α were respectively selected as $\{2, 1.5, 0.5, 0.5, 0.5\}$ and $\{0.35, 0.35, 0.20, 0.15, 0.15\}$ for FIR, IIR, convolution filter, FFT and DCT benchmarks. This hyper-parameter assignment allows the search to reach the maximum number of approximations under the predefined quality constraint. The parameter C provides a balance between exploration and exploitation of the search tree and can be experimentally found and tuned to get a trade-off between finding new search paths or to go deeper in the existing paths [65]. The parameter α controls the number of children nodes at each level. Higher values of alpha would allow the addition of more children nodes resulting in more approximate candidates to choose from

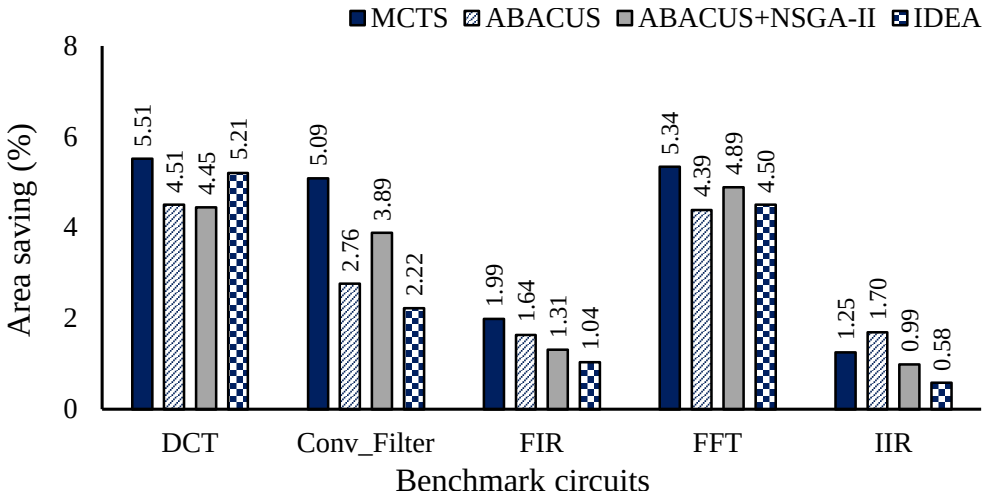
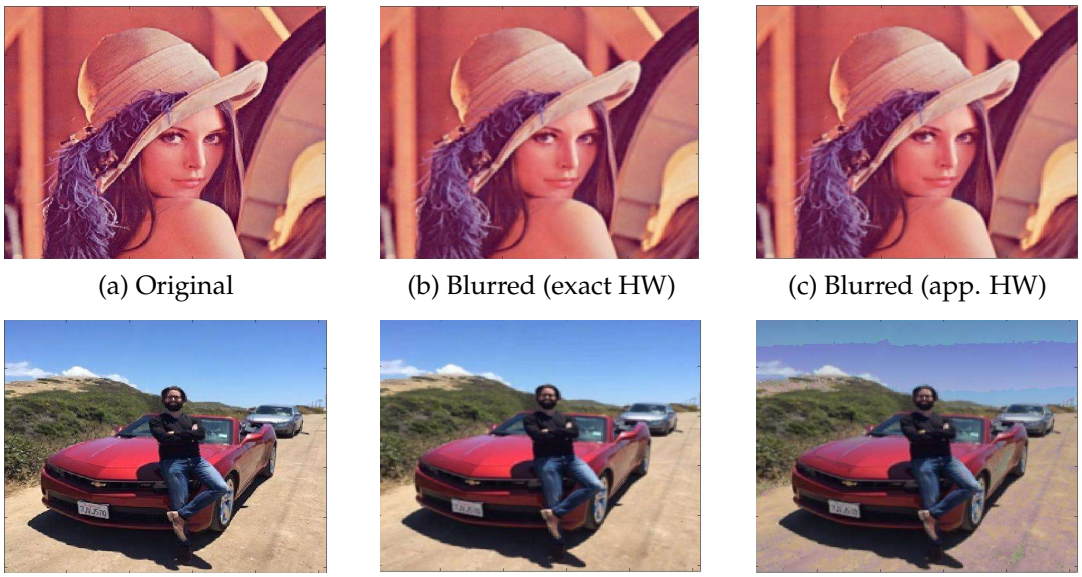


Figure 3.4: Area savings comparison for various benchmarks using the proposed framework.

in case the computational budget is higher. Lower values, however, can be set to achieve the same approximation level under a tight computational budget. For example, we observed that a higher value of α such as 0.45 would force the MCTS to add large number of branches to the search tree and the MCTS will not be able to go deeper in the search tree. On the other hand, a smaller value (such as $\alpha = 0.05$) would make the MCTS go deeper quickly in the search tree but reduce the chance of adding more approximation branches.

The hyper parameters of the other techniques were also selected to achieve the best results in terms of both the maximum number of possible approximate transformations and energy/area savings. For example, we used 25 generations using ABACUS for the FIR circuit, which were then followed by 40 approximate transformations in each generation. The results then were selected among the obtained variants with best energy and area saving. Our proposed framework presents better energy savings than others by reaching 32.08%, 61.69%, 11.46%, 24.97%, and 9.65% savings for DCT, convolution filter, FIR, FFT, and IIR, respectively. Figure 3.4 also represents the area saving of each technique over the five benchmarks. Our technique outperforms the others by reaching up to 5.51%, 5.09%, 1.99%, 5.34%, and 1.25% area reduction for DCT, convolution filter, FIR, FFT, and IIR, respectively.

Figure 3.5 also shows two representative samples from our training set (first row) and testing set (second row) used for evaluating the approximated convolution filter. Here, the filter was configured to generate blurry images. Figure 3.5-a, and d represent the original images given to the filter. Figure 3.5-b, and e represent the obtained outputs from the original hardware filter. Both cases represent a high PSNR value ($>$ 60 dB) related to tiny truncations of input data since the filter exploits a 32-bit fixed point (24 bits for fractional parts) number format. Figure 3.5-c, as a member of training set, just shows the PSNR value of 42.01 dB for the obtained filter with 29 approximations. Figure 3.5-f also depicts a sample of the test set which results in the



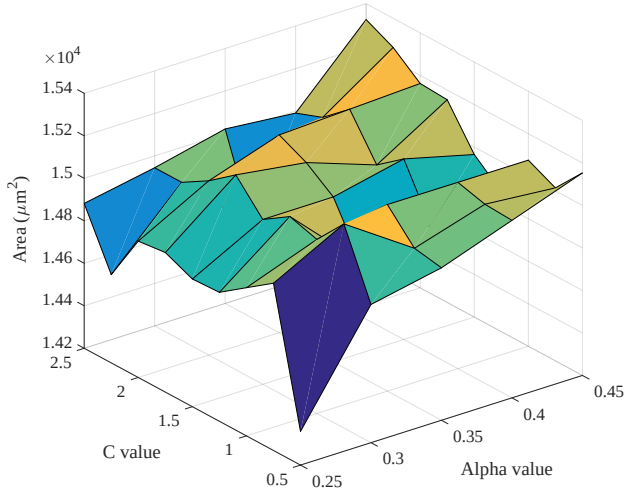
(a) Original (b) Blurred (exact HW) (c) Blurred (app. HW)

(d) Original (e) Blurred (exact HW) (f) Blurred (app. HW)

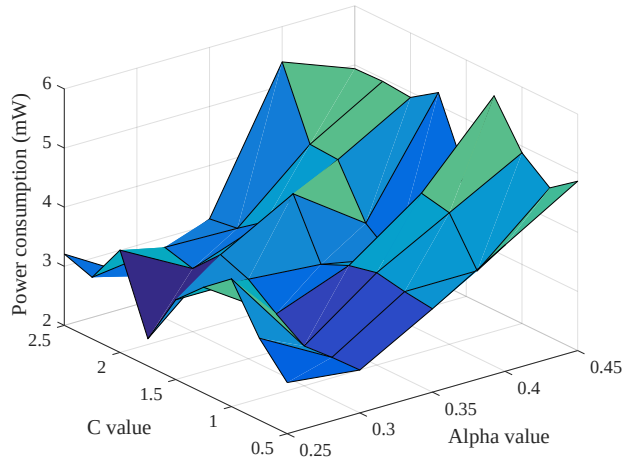
Figure 3.5: Samples of images from the training (a-c) and testing (d-f) sets for the convolution filter benchmark.

PSNR value of 25.32 dB.

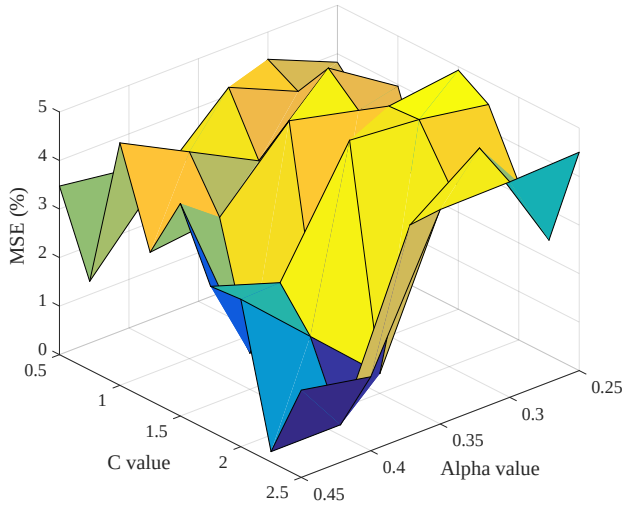
We also experimentally evaluated the impact of the framework hyper-parameters on the quality (i.e., error) and the characteristics (i.e., energy consumption and area) of generated AxACs. For this purpose, we performed experiments with different values for expansion parameter (α) and exploration constant (C). For each pair of (α, C), we performed AxAC synthesis for FIR benchmark with the budget of 10,000 iterations. Among the generated circuits, the one with higher number of approximate transforms, and more power/area saving in the design space was selected as the outcome of the framework. Figure 3.6 shows the impact of these parameters on the generated circuits. A larger value of the α (selected form $[0, 1]$) causes the search engine to spend more budget to find a better approximate transform for each circuit node. Therefore, the search budget may exhaust before visiting a large portion of nodes that could possibly be approximated. Besides, the high value of the C parameter forces the framework to increase the number of searches for finding the optimal transformation of the nodes visited few times so far. For α=0.45 only 17 transformations among 25 possibilities can be reached even with the minimum C value of 0.5 (Figure 3.6-a). That is why the design space is dedicated to more precise circuits with lower power and area savings (Figure 3.6-b and c). Alternatively, a smaller value for the parameter α forces the framework to find approximate transformations for higher number of circuit nodes with just a small budget. Consequently, more power and area savings can be obtained at the cost of a larger precision loss. The designer then can trade off among quality, optimization budget, and circuit characteristics by tuning these parameters.



(a) Area



(b) Power



(c) MSE

Figure 3.6: Impact of hyper-parameters (α and C) on area, power and precision of FIR benchmark circuit.

3.3 Towards a modular and flexible framework CIRCA

This section presents a joint work on CIRCA — an open-source framework — done in collaboration with other colleagues of computer engineering group of Paderborn university namely: Linus Witschen, Tobias Wiersema, and Hassan Ghazemzadeh Mohammadi. The contributions made by this thesis in the joint work of CIRCA framework are given below:

- This work contributed on the discussions leading to the concept of an open-source, extensible framework for synthesis of AxACs that could facilitate evaluation of different approximation and validation methods.
- This work contributed on the integration of the MCTS search method inside CIRCA.
- This work evaluated MCTS search on CIRCA and compared with two other search methods, and with two different approximation methods.

CIRCA is an open-source framework for rapid development and evaluation of AxAC synthesis techniques. CIRCA is modular and flexible so that new approximation, search and testing/verification techniques can be rapidly prototyped and evaluated.

3.3.1 Motivation behind development of CIRCA

The analysis of related search-based frameworks in Chapter 2 and the attempt to categorize them has shown that all these approaches have been developed for specific accelerator circuit types and limited approximation techniques. In particular, circuit generation is typically described as a monolithic block with interwoven phases for approximation, search, and assuring quality. Moreover, only few frameworks are openly available for experimentation. This situation severely hampers the development and evaluation of new techniques for approximating accelerator circuits, and the comparison to existing ones.

The proposed framework CIRCA aims to overcome these limitations and provides a flexible environment for AxAC generation. CIRCA is an attempt to provide a whole set of features that were missing from the existing frameworks. In particular, CIRCA fulfills the following technical requirements:

- *General:* The framework is not restricted to certain circuit types, error metrics, approximation and search techniques, or specific target technologies.
- *Modular:* The framework architecture enables the exchange of certain processing steps without affecting other steps.

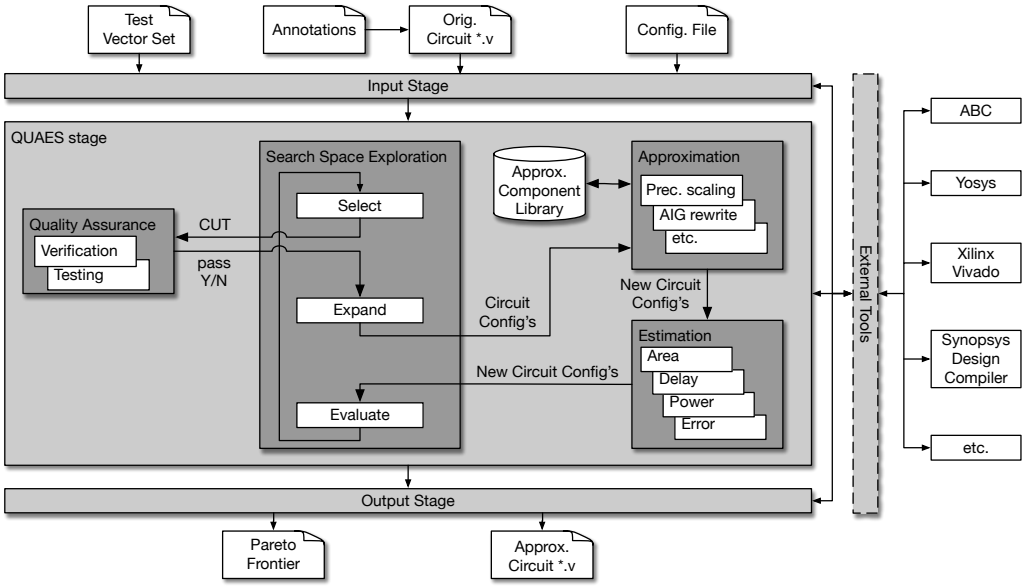


Figure 3.7: Architecture of the CIRCA approximation framework.

- *Compatible:* The framework, in particular its inputs and outputs, connect to other, widely used academic and commercial front-end and back-end tools, e.g., tools synthesizing accelerator circuits for ASIC or FPGA technology.
- *Extensible:* The framework facilitates the swift implementation and evaluation of new techniques.

Additionally, the framework satisfies the community requirement:

- *Open-source availability:* The framework is publicly available[1] and allows other researchers to use, modify, and extend it.

3.3.2 Concept and architecture of CIRCA

Figure 3.7 shows the architecture of the CIRCA framework. We have developed the architecture with the key features for a flexible approximation framework in mind, namely generality, modularity, extensibility, and compatibility. CIRCA divides into three stages: the input stage, the QUAES stage, and the output stage. The input and output stages frame the QUAES stage and prepare user-supplied circuits for approximation, report on the approximation results, and ensure compatibility between different front-end and back-end tools and the QUAES stage.

Input stage

The input stage fulfills two main tasks, preprocessing the input design for the approximation process and ensuring compatibility to external formats and tools. In preprocessing, the input stage has to identify a set of sub-circuits within the original

[1] CIRCA is available under: `https://go.upb.de/circa`

design which is amenable to approximations, e.g., arithmetic components such as adders and multipliers. This set of sub-circuits is denoted in CIRCA as *candidate set*. Suitable candidates can either be identified by a designer through code annotations in the original design (as indicated in Figure 3.7) or by automated methods. Furthermore, the input stage reads in the user-provided CIRCA configuration file that specifies the functionality of the stages and blocks as well as the test vector set that is required for testing-based quality assurance. The input stage also provides compatibility between CIRCA's approximation process in the QUAES stage and formats used by external tools, e.g., ABC [68] and Yosys [69].

The QUAES stage

The Quality Assurance, Approximation, Estimation, and Search Space Exploration (QUAES) stage is the main stage for generating AxACs, and is designed to implement iterative, search-based approximation approaches with different approximation techniques and both formal verification and testing for circuit validation. In the QUAES stage, the candidates are subjected to approximation and different approximated versions of the candidates will be generated. We denote an approximated version of a candidate as *variant* and the overall circuit with instantiated variants for the candidates as *circuit configuration* or *node*.

Through splitting QUAES into four independent blocks i.e., quality assurance, approximation, estimation, and search space exploration, CIRCA facilitates the clear distinction between tasks and allows the exchange of methods in one block without affecting the other blocks. Since CIRCA targets search-based approximation processes, *search space exploration* acts as the central control block of the QUAES stage, invoking the other blocks whenever necessary.

In each iteration of the search, the select step receives a set of circuit configurations, each with an annotated vector of estimated parameters, and selects the next configuration to be further considered, i.e., to be expanded. The selection procedure relies on a certain search heuristic. The expansion step generates possible approximation variants from the selected nodes through approximation block. The set of new nodes after expansion step are added to a list that keeps nodes that are yet to be validated. The evaluate block can then use the statistics provided by the estimation block to rank the nodes according to the heuristic criteria of the search algorithm. The search continues with the select step. The termination criteria of search can be specified according to the algorithm e.g., the greedy algorithm terminates when there are no more nodes providing less area than the current node.

The *quality assurance* block provides the validation of the circuit configuration, denoted as *Circuit-under-Test (CUT)*, sent from the select step of search. Here, either formal verification or testing can be employed to decide if the CUT satisfies the quality constraints and passes the quality assurance check. While approaches based on formal verification lead to conceptually much stronger statements about quality

than testing, they also tend to very long run-times. For testing, a set of test vectors can be provided to CIRCA via the input stage. Quality assurance procedures can then apply all these vectors or a randomly selected subset to a circuit-under-test (CUT). In the current state of CIRCA, testing has not been implemented yet, therefore the results presented in Section 3.3.5 were obtained with formal verification as the only available validation technique. The testing component however is the part of the CIRCA concept and is planned to be implemented in future. For formal verification, we use an approximation miter like the one mentioned in [14]. Following the terminology of [17], we denote the approximation miter as sequential quality constraint circuit (SQCC). The quality constraints are provided by the user as inputs to CIRCA and are defined by appropriate error metrics with corresponding error thresholds. If the CUT fails the quality assurance check, the select procedure will, depending on the employed search algorithm, either abort the search or pick the next best circuit configuration for validation. In the latter case, the search terminates if there are no more valid configurations. If the CUT passes validation, the expand procedure grows the search space by calling the approximation block to create new circuit configurations by applying certain approximation techniques.

The *approximation* block receives a set of circuit configurations with candidates indicated for approximations. CIRCA allows multiple approximation techniques to be employed during the approximation process. Currently, two approximation techniques are implemented: precision scaling (PS) and approximation-aware AIG rewriting (AIG) [14]. Both AIG, carefully following the descriptions in [14], and PS have been implemented into the widely used ABC tool [68] to have both techniques available in a tool which provides a broad range of synthesis, optimization, and verification functionality. The approximation can also access a component library with approximated sub-circuits, which is beneficial for two reasons: First, it is rather likely that one circuit component will be approximated for several times. This happens when the overall circuit contains identical candidates, e.g., multiple occurrences of an 8-bit unsigned adder. Storing the approximated versions of such components, i.e., generating a component library on-the-fly, and retrieving them the next time can greatly save computations. Second, CIRCA can leverage already existing libraries of approximated components, e.g., [38, 46]. Such libraries could be provided as an input to CIRCA by the user.

The circuit configurations are then passed on to the *estimation* block and, at the same time, the approximated variants for the candidates are stored in a library of approximated components for later re-use. The estimation block receives a list of circuit configurations with newly approximated component variants and estimated target metrics of interest. Currently, CIRCA uses ABC's *if* command to estimate area parameters for FPGA 4-LUT architectures. The parameters are stored in the circuit object and can be used for further processing. The evaluation step determines estimated values for parameters of interest. Typically, these parameters cover metrics related to area, delay, and power consumption but can also include estimates for the

error metrics.

The output stage

The output stage performs post-processing on these circuit configurations. Depending on the CIRCA configuration, the output stage either returns the best approximated circuit, which is the circuit that respects the error constraints and minimizes some user-defined parameter such as area or energy, or provides an approximated Pareto-filtered set of approximated circuits for further analysis and consideration. The output stage also connects to back-end synthesis tools for actual circuit implementation, such as Synopsis Design Compiler for ASIC technology or Xilinx Vivado for FPGA implementation.

3.3.3 Search space exploration methods in CIRCA

The modular structure of CIRCA allows developers to implement any heuristic search algorithm using the abstract classes and method provided in the current implementation. Currently, the CIRCA framework includes three search methods: *hill climbing* (HC), *simulated annealing* (SA), and *Monte Carlo tree search* (MCTS).

Hill climbing (HC)

Greedy hill climbing (HC) has been used as one of the most common search methods to solve optimization problems [70, 71]. In CIRCA, HC is implemented as a greedy method to minimize hardware area. Starting from an initial point in the search space — in our case the original circuit *root* — HC iteratively moves in the direction of the most negative difference, i.e., the decisions are only depending on the current state. In case the most negative node is found to be invalid by the validation block, it picks the next best node until it finds a valid node to expand. The expand step then generates all children nodes for the selected node and perform approximation. The estimate step then provides parameters of interest for each node such as area and power. HC then reiterates from the select step. HC terminates when no more valid nodes are available to select from.

An obvious drawback for HC is that the search terminates once it reaches a local minimum. Thus, it is not guaranteed to find a global minimum.

Simulated annealing (SA)

Simulated annealing (SA) has been adapted from a phenomena from matter physics involving heating and controlled cooling of a material [72]. SA has been used as a probabilistic search method to solve combinatorial optimization problems by approximating a global optimum under a given runtime constraint. In each iteration, first, a random node from the available nodes is selected in the select step. Then,

the acceptance of the selected node is determined in the following way: A node improving the target metric is always accepted for selection. Nodes which worsen the target metric might also be selected to allow for the exploration of the search space. A worse node is accepted under the criteria:

$$random[0,1) < e^{-\frac{\Delta D}{T}}. \tag{3.5}$$

T is the current simulated temperature, and ΔD is the change in the target metric, where ΔD is positive for a worse new node.

Initially, the simulated temperature T is high, i.e., worse nodes are more likely to be accepted. During the search, however, the temperature cools down at the rate α. The temperature is updated after reaching thermal *equilibrium* ξ at each temperature. A lower temperature leads to a lower probability of worse nodes being selected, i.e., SA moves from exploring the search space at the beginning of the search to moving towards better solutions at the end of the search. Once the temperature T_{Min} is reached, SA terminates.

We assume that increased error constraints lead to larger savings in the target metric hardware area. Thus, in the expand step, only the children of the current node are considered in the open-list which guides the search in the direction of lower quality constraints or higher error constraints, respectively. The search space is pruned by prohibiting moving towards the parent nodes; thus, avoiding a search space explosion.

Monte Carlo tree search (MCTS)

MCTS is an intelligent search algorithm that incrementally builds a search tree with a balanced selection policy. The current versions of MCTS algorithms, which are widely used in domains such as games, require a slight modification to be fitted for the domain of AxAC synthesis (Section 3.2). For instance, the simulation time to validate the quality of a circuit instance is very long in comparison with the other domains[2]. This prohibits the possibility of evaluating the current search node by knowing the quality of a subset of probable future nodes originating from here. Consequently, the evaluation of each node is performed independently.

The detailed steps of our proposed MCTS are represented in Algorithm 2. We incrementally form a search tree in which a branch represents an approximate transformation and a node specifies an approximate circuit instance, as before. In the *Selection* step the tree is always traversed from the root node, selecting the node with the maximum *Upper Confidence Bound applied to Trees* (UCT) value in each level (recall the UCT formulation from Section 3.2).

[2]For example, a playout in Computer-*Go* took just a few micro seconds while the validation of a small size circuit like FIR could take up to a couple of minutes.

Algorithm 2: MCTS algorithm in CIRCA framework

```
   Input: O= Verilog description of the accelerator circuit , M= Simulation budget ,
   ε_max = pre-defined quality threshold
   Output: A={Approximated instances satisfying quality constraints}
1  while M > 0 do
       /* Selection step                                                    */
2      if isEmpty( open_list ) then return A;
3      cn ← root;
4      while cn.expanded do
5          if isEmpty( getActiveChildren( cn )) then
6              backpropagate( cn, Nreward ); // Dead end
7              break;
8          else
9              cn ← UCTSelect( getActiveChildren( cn ));
10     if !isExpandableNode( cn ) then continue;
11     closed_list.add( cn );
12     open_list.remove( cn );
13
       /* Validation step                                                   */
14     cn.valid ← quality_assurance.validate( cn, ε_max );
15     if (cn.valid) then
16         A.add( cn );
17
           /* Expansion step                                                */
18         children ← generateChildren( cn );
19         filter( children, open_list );
20         filter( children, closed_list );
21         cn.children ← children;
22         open_list.add(children);
23         cn.expanded ← true;
24
25         approximation.approximate( children );
26         estimation.estimate( children );
27
           /* Evaluation step                                               */
28         reward ← computeAreaSaving( cn );
29         backpropagate( cn, reward );
30     else
31         cn.active ← false;
32         backpropagate( cn, Nreward );
33 return A;
```

Next, the quality of the selected node is examined in the *Validation* step to assure that the required precision is not violated. Otherwise, a *negative reward* is again back-propagated from the current node to the root. If the quality constraint is satisfied, the *Expansion* step generates the children of the node. Finally, the reward of the current selected node is calculated in the *Evaluation* step by taking into account the savings in the target metric for the current node. This reward value is then back-propagated up to the root.

Figure 3.8 gives a complete view of how the MCTS algorithm has been implemented inside the CIRCA framework. The MCTS block of Figure 3.8 is explained

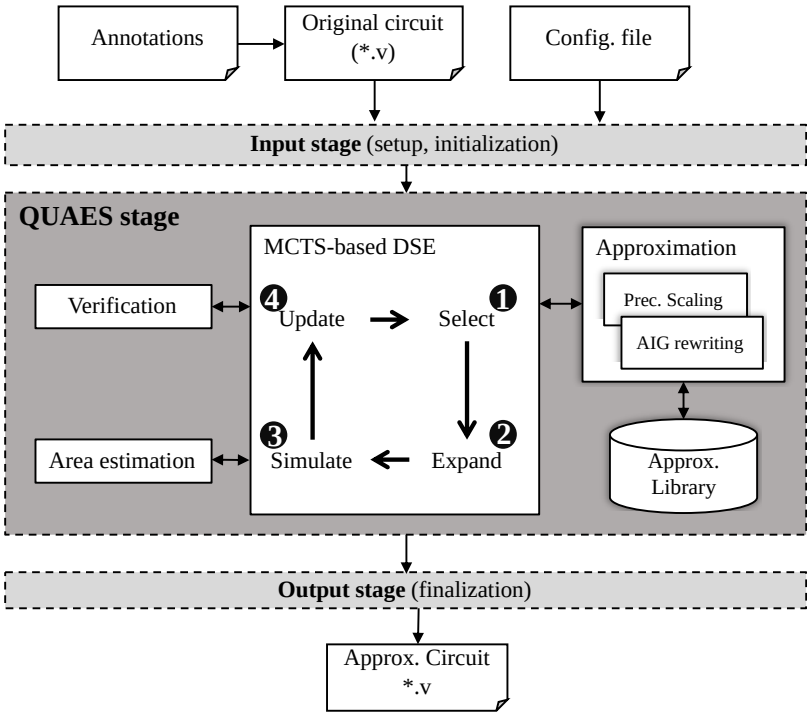


Figure 3.8: MCTS flow in CIRCA approximation framework.

in Figure 3.9. Here, the root node represents the target circuit that has only three components that can be approximated. Each child represents a circuit configuration with one stride from the root. For example, the child with label $[1, 0, 0]$ represents a circuit instance with one approximate transformation (e.g., 1 least significant bit truncation) on its first component. In every iteration of MCTS, an approximate candidate is selected and evaluated in term of precision quality. If the quality check is satisfied the node is labeled as a valid node (green) otherwise as a dead node (black striped). Next, the possible children of the current valid node are added to the search tree for future iteration. Then, the benefit of the applied approximation on the current node is calculated as a reward value and back-propagated to the root node. This process iterates until the simulation budget of the search is exhausted. Finally, the best nodes within the tree, i.e., with minimum target metric, are reported.

3.3.4 Bounding the search space

To allow for a search space expansion in a controlled manner, CIRCA differentiates between global and *local quality constraints*. Global quality constraints apply to the circuit configurations. Local quality constraints are constraints on error metrics and apply to generated variants of circuit components, e.g., the worst-case error of the variant can be bounded. In CIRCA this happens in the approximation block. As a result, in the course of the approximation process the variants created for one candidate will exhibit a step-wise decrease in accuracy. Naturally, these local quality constraints do not directly relate to the overall circuit's quality, which is checked in the quality assurance block.

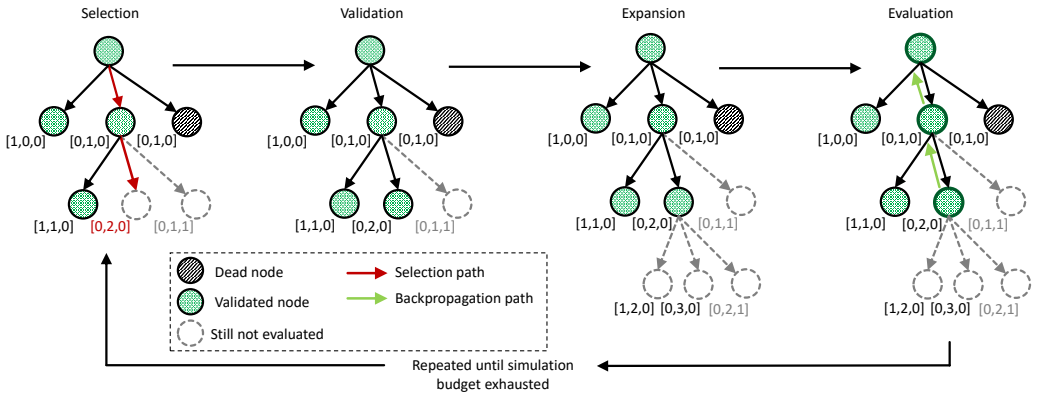


Figure 3.9: Major steps in the MCTS algorithm.

The user can bound the search space by controlling its density and dimensionality. The density is given by the distance between directly neighboring nodes in the search space. CIRCA allows the user to determine the density by setting step sizes for the local quality constraints. For example, a step size of 0.1% for the local error metric worst-case error will populate the search space with doubled density compared to a step size of 0.2%

Furthermore, in each iteration of the search process CIRCA approximates n candidates $C_0, \ldots, C_{n-1}$ of a circuit configuration by applying to each candidate C_i a number of A_{C_i} different approximation techniques and checking for E_{C_i} different local error metrics. These parameters span the dimensions of the search problem. Overall, the total number of dimensions D of the search space is given by Equation 3.6.

$$D = \sum_{i=0}^{n-1} A_{C_i} \times E_{C_i} \tag{3.6}$$

By setting appropriate values for n, A_{C_i} and E_{C_i} the user limits the complexity of the search problem. For example, for the experiments presented in Section 3.3.5 we use $A_{C_i} = E_{C_i} = 1 \;\; \forall i = 1 \ldots n$ with n ranging between 2 and 23, as denoted in Table 3.2.

3.3.5 Experimental setup and results

CIRCA is an open-source software and mainly being coded in Python, following the style guide for Python, PEP-8 [73]. The input stage, the output stage, and the QUAES' processing blocks (quality assurance, approximation, estimation, and search space exploration) are implemented as classes or abstract base classes. During the setup phase, information from the configuration file is either used to set class attributes accordingly or to instantiate the appropriate subclass for the abstract base class. The concept of abstract base classes and abstract methods is used in CIRCA

Table 3.2: Sequential benchmark circuits

Benchmark	Bit-width	# Candidates	Area[§]	QoR
butterfly [74]	32[†]	7	19 038	WC error (%)
fir_8tap	67	15	12 401	WC error (%)
fir_pipe_16 [75]	18	23	8935	WC error (%)
pipeline_add [76]	40	2	572	WC error (%)
rgb2ycbcr [77]	24[‡]	5	4981	WC error (%)
ternary_sum_nine [76]	20	4	1484	WC error (%)
weight_calculator	12	4	2272	WC error (%)

[§] Number of 4-LUTs after mapping with ABC.
[†] Concatenation of real and imaginary part.
[‡] Concatenation of three channels, each 8-bit wide.

to guide developers through the implementation of new methods and to highlight methods required by CIRCA's approximation flow explicitly.

Setup of experiments

For experimentation, we have selected seven sequential circuits (Table 3.2) from our compiled publicly available benchmark suite PaderBench[3] and manually annotated adder and multiplier components in the data path as approximation candidates. CIRCA has been set up to vary the worst-case error bound from 0.25% to 5.0%, expressed in percentage of the circuit's maximum possible output value, to employ formal verification with ABC's *dprove* for assuring quality and the hardware area as target metric. Moreover, we have systematically experimented with all three search methods HC, SA, and MCTS in different parametrization and with precision scaling (PS) and *And-Inverter-Graph rewriting* (AIG rewriting) as approximation techniques.

We have run the approximation flow five times for each benchmark circuit and determined the averages as representative results. The experiments have been performed on a compute cluster which runs Scientific Linux 7.2 (Nitrogen), comprises nodes with an Intel® Xeon E5-2670@2.6GHz (16 cores) and from 64 up to 256 Gigabyte main memory, of which it provides 2 Gigabyte main memory and one core per job, i.e., single Benchmark run.

Table 3.2 elaborates on the characteristics of these benchmark circuits, which are the output bit widths, the hardware area in terms of number of used FPGA 4-LUTs as reported by ABC [68], and the number of manually identified candidates for approximation. The selected sequential circuits represent accelerators from various application domains ranging from small arithmetic blocks to complex data-paths and are briefly described as follows:

[3] `http://go.upb.de/paderbench`

- *butterfly* is the basic operation in the FFT/DFT, which is used in many signal processing applications. It takes two 32-bit wide complex numbers and a 32-bit twiddle factor as input and returns the result of computation as a 32-bit complex number [74].
- *fir_8tap* implements an eight-tap Gaussian low-pass FIR filter, which is commonly used in the domain of signal processing. It takes a 32-bit wide data input and provides a 67-bit wide output.
- *fir_pipe_16* is a 16-tap FIR filter [75] that uses pipelined blocks to increase the throughput. Both the input and output are 18-bit values.
- *pipeline_add* [76] is a pipelined adder circuit that takes two 20-bit numbers as input and gives a 20-bit sum as output.
- *rgb2ycbcr* is a module in the JPEG encoder [77] that performs a color space transformation. It takes a 24-bit RGB value as input and provides a 24-bit wide output which represents the YCbCr color space value.
- *ternary_sum_nine* is an adder tree [76] that performs an addition of nine binary words using four adder chains. Its input comprises nine values to be added, each 16-bit wide, and the output is a 20-bit value containing the sum.
- *weight_calculator* is a simplified version of an industrial scale weight calculator which is used to control the hoppers of a multi-head weigher. It reads an input value stored in RAM and provides outputs to control the hoppers. For verification purposes a 12-bit output that represents the combined weight of the hoppers is used in our experiments.

Results

We evaluated the impact of the search and the approximation methods on the mentioned benchmarks. For fair comparison, multiple configurations of the search methods were generated through setting different values of parameters. The parameters include for example the exploration parameter of MCTS (C) and *Equilibrium* for SA method among others. The area saving results presented in this section represent the best results obtained for a search method among all configurations. Here we report the area saving results separately for two different approximation methods employed i.e., PS and AIG. Furthermore, for current set of experiments, formal verification was used as a validation technique. In future, it is planned that a testing-based validation will be added to CIRCA.

Figure 3.10 displays the minimum area obtained normalized to the area of the original circuit over the worst-case error bounds using the PS method. Similarly, in Figure 3.11 we show the minimum area obtained normalized to the original for same benchmarks over the worst-case error bounds using the AIG method. Comparing the approximation techniques over the experiments, we see that PS could achieve

area savings of up to ≈55 %, while AIG could only achieve area savings of up to ≈25 %. An explanation for the superiority of PS over AIG in our experiments is that we have selected arithmetic components as candidates. For such components, PS degrades the accuracy more gracefully than AIG. AIG targets nodes on the critical path, which usually affects the carry-chain of a multiplier or adder. Approximating such nodes leads to large errors and, in turn, to the rejection of the variant if moderate error bounds are applied. PS introduces smaller errors when operating on the least significant bits of the output vector of a candidate and thus approaches the error boundaries more carefully.

Figure 3.10 and 3.11 also reveal the impact of employed search method on the quality of the approximated outcome. Differences in the results can be observed among the same search methods when parametrized differently, and also among the different search methods.

For SA, however, increasing the number of allowed iterations does not lead to significant differences in area savings. On the one hand, this could be caused by our implementation's method for pruning off the search space. On the other hand, fine-tuning the search parameter might lead to further improvements. However, in general SA performs very well and achieves higher savings than HC — especially, for the benchmarks *pipeline_add* and *rgb2ycbcr*.

MCTS performance lies between HC and SA. For an improved performance, MCTS needs to compute more iterations. This will allow MCTS to explore more branches of the search tree to find better solutions. With higher budgets, MCTS will also be able to trade off between exploitation and exploration. For example, setting the exploration constant (C) to a higher value will cause MCTS to explore branches with low visit counts more often, whereas setting C to a low value will result in MCTS exploiting nodes which proved to be more rewarding in the previous iterations.

Overall, the experiments show that the quality of the result highly depends on the approximation technique as well as on the employed search method. This underlines the necessity of conducting extensive experiments with different approximation and search techniques, which is well supported by CIRCA. Furthermore, our results also point to the fact that the achievable area savings strongly depend on the input design. Some benchmarks, e.g., $butterfly$ or fir_pipe_16, describe challenging approximation problems for which all search methods could achieve only very small area savings.

In Table 3.3 and 3.4, we list the average run-times and the average number of performed iterations of the employed search methods, again for the PS and AIG methods, respectively. For instance, the entry for the benchmark butterfly under the error bound of 0.25 % for MCTS elaborates that the particular experiment took three hours 58 minutes and 24 seconds on average and performed a total of 95 iterations. The formal verification employed in the quality assurance block has been identified as the dominating part in the runtime of the approximation flow. The run-times of

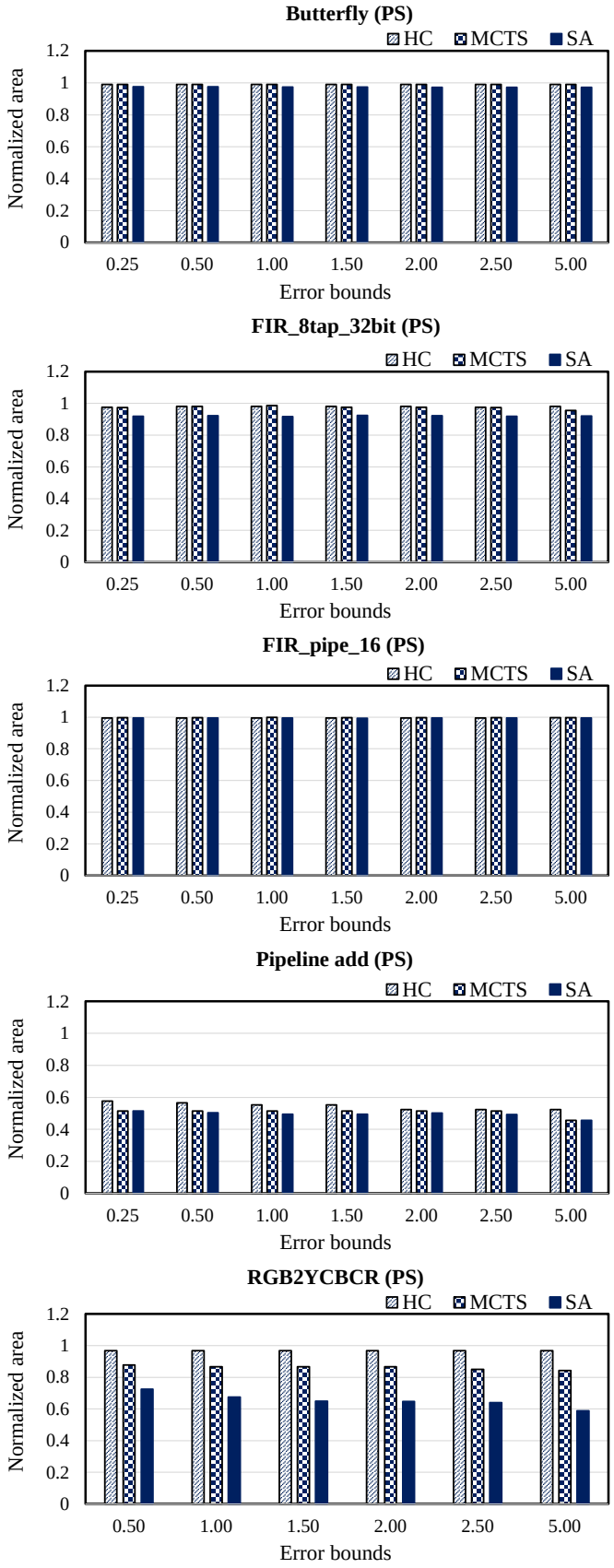


Figure 3.10: Area savings results for different benchmarks using precision scaling technique against different error bounds with CIRCA framework (continued on next page).

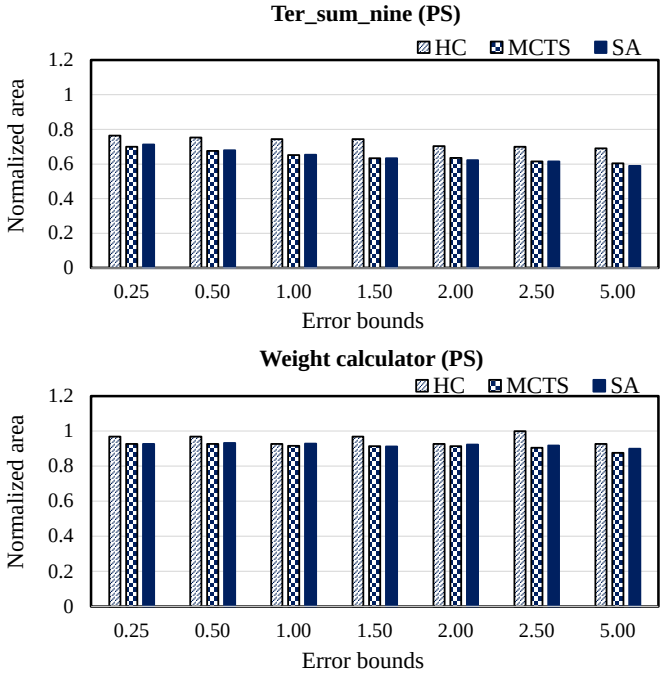


Figure 3.10: Area savings results for different benchmarks using precision scaling technique against different error bounds with CIRCA framework (continued from previous page).

the approximation processes range from a few seconds up to several days, depending on the number of verifications performed and depending on the complexity of the occurring verification problems. However, due to randomness in the taken path through the search space as well as in the applied approximations, the complexity of the occurring verification problems may differ. Thus, the runtime of an approximation process may vary even though the same number of verifications has been performed for the same benchmark circuit, e.g., the *butterfly* benchmark for HC method.

Comparing the run-times of the three search methods reveals that HC often provides significantly shorter run-times than SA or MCTS. However, as Figures 3.10 and 3.11 show, the area savings were lower on average as well. Furthermore, HC performs, in general, significantly less iterations compared to SA and MCTS. There are two explanations to this: The first relates to the way we count iterations. Basically, we increase the iteration count once a node gets selected. Compared to HC, SA tends to accept more nodes which naturally increases its iteration count. MCTS performs an iteration when performing back propagation. This leads to more iterations, although the runtime is not increased significantly. Second, HC might get stuck in a local minimum quickly. While HC then terminates the search, SA and MCTS continue with more iterations.

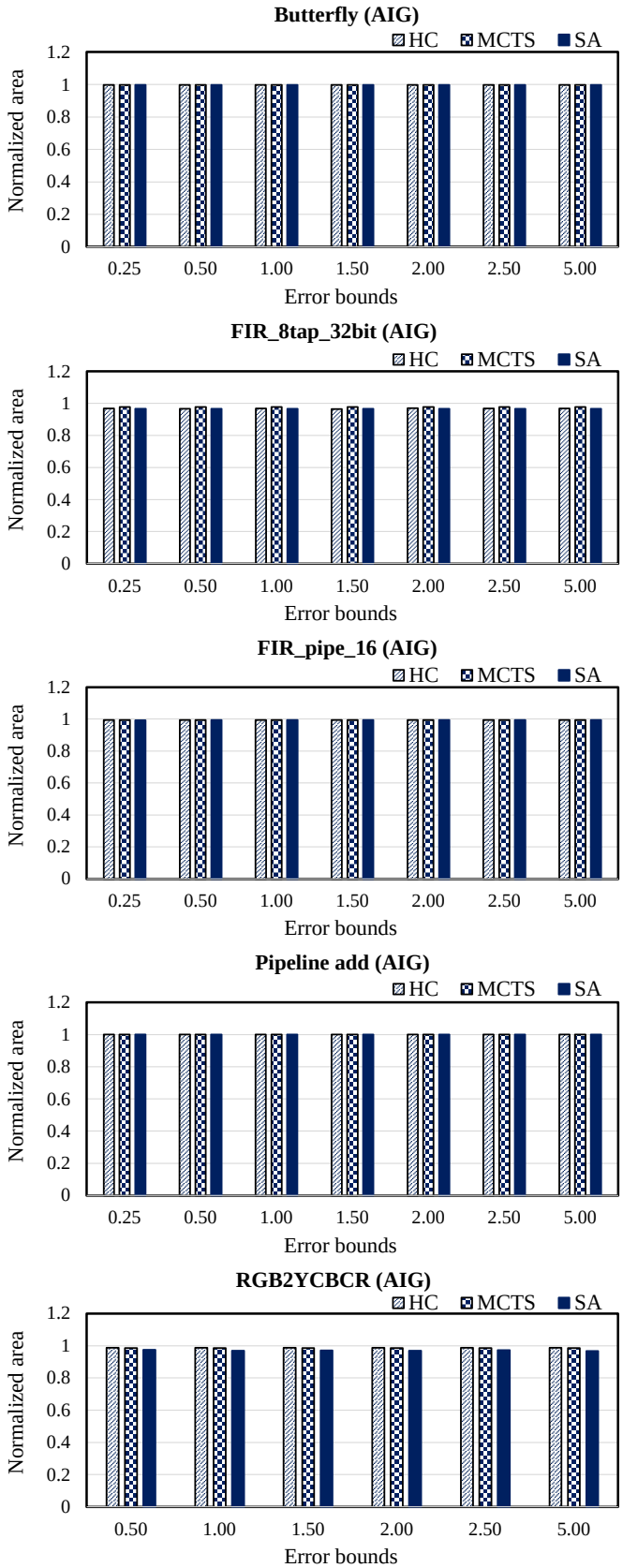


Figure 3.11: Area savings results for different benchmarks using AIG-rewriting technique against different error bounds with CIRCA framework (continued on next page).

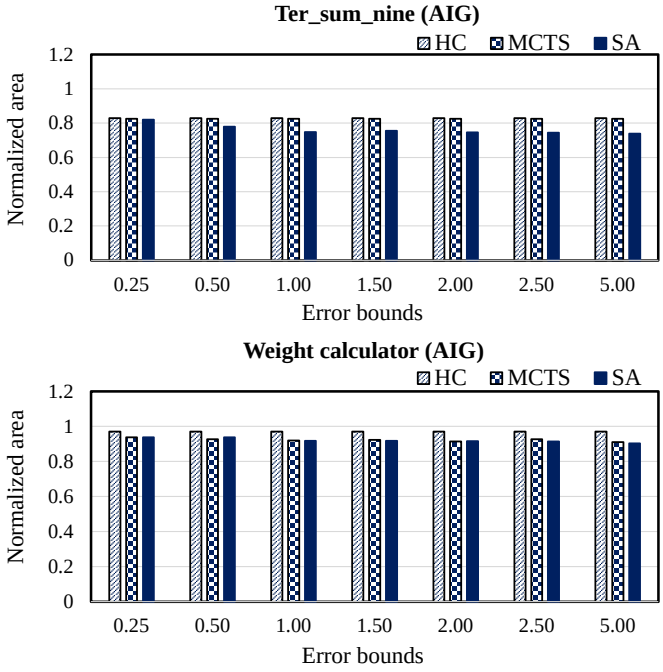


Figure 3.11: Area savings results for different benchmarks using AIG-rewriting technique against different error bounds with CIRCA framework (continued from previous page).

3.3.6 Further Discussion

The main purpose of this section is to explain the differences of two setups of experiments provided in this chapter so far. First setup is explained in Section 3.2 and was part of our proposed MCTS-based framework whereas the second setup is the open-source framework CIRCA and it is presented in Section 3.3. Moreover, at the end of this section, we elaborate the role of setting up a reasonable computational budget for MCTS and show with experimental results that how it affects the performance metrics.

In MCTS-based framework, we have utilized a wide range of approximation techniques to generate approximate modules later to be used in the form of an approximate transformation library. These library components were obtained with variety of approximation techniques such as carry chain cutting, truncation, etc. Moreover, approximate transformations were randomly selected with a weighted heuristics based on their impact on area and power consumption. The MCTS-based framework was based on a testing approach to validate the intermediate nodes and therefore it was possible to allocate larger (up to 100$\times$ more than allocated in case of CIRCA) computational budget to MCTS.

The CIRCA framework on the other hand puts a number of limitations that in turn do not well align with how MCTS grows the search tree. For one, the CIRCA framework was based on a formal verification flow for validating the intermediate approximate variants which is an extremely time consuming task and would not allow MCTS to complete a reasonably larger number of iterations. Second, CIRCA

Table 3.3: Average runtimes of the CIRCA framework and number of selected nodes (PS).

Benchmark	Worst-case error bound [%] 0.25		0.50		1.0		1.50		2.00		2.50		5.00	
HC (Greedy)														
butterfly	40:28	(4)	45:16	(4)	49:10	(4)	1:00:21	(4)	49:49	(4)	39:46	(4)	46:12	(4)
fir_8tap_32bit	9:25:14	(5)	7:33:10	(4)	7:49:28	(4)	7:32:53	(4)	7:32:53	(4)	8:33:10	(5)	7:33:13	(4)
fir_pipe_16	45:39	(2)	55:04	(2)	59:15	(2)	42:41	(2)	30:47	(2)	40:10	(2)	41:36	(2)
pipeline_add	14:13	(16)	1:12	(16)	7:02	(16)	1:13	(16)	1:13	(16)	1:12	(16)	1:11	(16)
rgb2ycbcr			15:03	(6)	41:24	(6)	1:17:45	(6)	24:29	(6)	1:14:19	(6)	28:21	(6)
ter_sum_nine	21:46	(21)	3:06	(22)	24:06	(23)	19:16	(23)	25:15	(26)	9:32	(27)	16:22	(28)
weight_calculator	7:20:39	(3)	7:22:14	(3)	31:23	(3)	10:51:38	(3)	24:49	(3)	7:10:31	(2)	21:34	(3)
MCTS														
butterfly	3:58:24	(95)	3:53:00	(95)	3:49:10	(95)	3:47:56	(95)	3:50:27	(95)	3:52:31	(95)	3:52:07	(95)
fir_8tap_32bit	8:08:06:50	(100)	8:04:20:57	(100)	7:13:27:07	(100)	4:17:34:27	(100)	4:04:28:28	(100)	8:11:35:56	(100)	1:08:39:45	(100)
fir_pipe_16	3:35:23	(100)	3:37:37	(100)	3:35:11	(100)	3:34:55	(100)	3:47:30	(100)	3:34:41	(100)	3:38:55	(100)
pipeline_add	5:45	(84)	5:47	(85)	6:00	(88)	6:14	(88)	5:37	(88)	5:37	(88)	5:27	(84)
rgb2ycbcr			3:06:07	(100)	3:38:02	(100)	3:52:11	(100)	3:57:49	(100)	4:40:35	(100)	10:05:21	(100)
ter_sum_nine	11:06	(100)	10:41	(100)	10:40	(100)	10:45	(100)	10:50	(100)	10:49	(100)	10:51	(100)
weight_calculator	2:04:15:33	(42)	2:01:20:45	(33)	7:10:31:40	(91)	5:04:19:12	(85)	4:22:05:15	(91)	3:08:29:28	(75)	4:15:08:42	(99)
SA														
butterfly	8:07:57	(111)	10:06:59	(113)	8:31:36	(120)	8:18:53	(114)	9:45:50	(118)	8:32:58	(114)	10:33:04	(123)
fir_8tap_32bit	3:17:55:22	(66)	2:18:02:27	(66)	1:23:39:57	(66)	3:02:25:42	(66)	2:03:17:33	(66)	2:15:55:44	(66)	1:16:37:38	(66)
fir_pipe_16	12:28:46	(141)	10:58:15	(144)	12:50:16	(146)	13:40:44	(148)	14:50:37	(150)	12:43:25	(145)	12:40:44	(145)
pipeline_add	38:47	(40)	37:05	(39)	41:56	(37)	25:24	(40)	23:33	(39)	25:28	(39)	14:27	(40)
rgb2ycbcr			1:54:40	(84)	2:06:58	(88)	3:29:51	(93)	3:09:43	(94)	3:22:57	(93)	5:38:05	(100)
ter_sum_nine	58:54	(44)	41:25	(50)	41:00	(51)	59:46	(57)	1:08:23	(58)	57:51	(62)	1:04:31	(63)
weight_calculator	11:35:57	(8)	22:41:51	(9)	1:15:25:13	(14)	1:11:01:40	(15)	1:08:19:58	(17)	1:14:17:34	(17)	1:04:45:36	(19)

The runtimes are shown in the format *days:hours:minutes:seconds* and are followed by the average number of performed iterations.
† These benchmarks exceeded the time limit.

Table 3.4: Average runtimes of the CIRCA framework and number of selected nodes (using AIG rewriting).

Benchmark	Worst-case error bound [%] 0.25		0.50		1.0		1.50		2.00		2.50		5.00	
HC (Greedy)														
butterfly	1:02:21	(4)	52:47	(4)	50:24	(4)	57:58	(4)	45:47	(4)	34:08	(4)	42:34	(4)
fir_8tap_32bit	1:34:51	(7)	1:38:54	(8)	1:25:54	(7)	1:51:05	(9)	1:22:59	(7)	1:46:13	(7)	1:33:10	(7)
fir_pipe_16	1:11:07	(4)	1:40:46	(4)	1:39:15	(4)	51:12	(4)	1:47:36	(4)	51:34	(4)	1:13:00	(4)
pipeline_add	7	(1)	7	(1)	7	(1)	53	(1)	7	(1)	7	(1)	1:03	(1)
rgb2ycbcr			1:46	(2)	1:47	(2)	1:50	(2)	6:16	(2)	4:34	(2)	1:49	(2)
ter_sum_nine	22:14	(14)	19:54	(14)	27:46	(14)	13:50	(14)	13:09	(14)	32:35	(14)	23:24	(14)
weight_calculator	13:11	(2)	10:14	(2)	12:55	(2)	17:30	(2)	10:09	(2)	10:16	(2)	13:23	(2)
MCTS														
butterfly	4:45:44	(97)	4:40:59	(97)	4:39:18	(97)	4:40:47	(97)	4:39:05	(97)	4:40:42	(97)	4:38:43	(97)
fir_8tap_32bit	4:57:40	(100)	0:04:54:59	(100)	4:58:34	(100)	4:58:39	(100)	4:59:34	(100)	4:58:52	(100)	5:00:21	(100)
fir_pipe_16	8:14:19	(100)	0:07:53:47	(100)	7:31:58	(100)	7:46:57	(100)	7:45:46	(100)	7:43:10	(100)	7:36:02	(100)
pipeline_add	2:36	(41)	2:27	(41)	2:40	(41)	2:30	(41)	2:43	(41)	2:33	(41)	2:54	(41)
rgb2ycbcr			2:07:58	(96)	1:39:34	(96)	1:36:00	(96)	1:43:52	(96)	1:36:35	(96)	1:25:30	(96)
ter_sum_nine	2:07:48	(96)	1:20:24	(96)	58:26	(96)	50:47	(96)	40:14	(96)	40:09	(96)	26:58	(96)
weight_calculator	5:15:12:14	(92)	5:04:41:30	(61)	3:03:58:10	(98)	3:00:19:07	(99)	1:22:26:16	(100)	2:21:57:53	(100)	1:08:43:10	(99)
SA														
butterfly	23:54:37	(270)	22:13:58	(270)	22:00:02	(270)	22:53:34	(270)	0:21:47:41	(270)	21:58:07	(270)	22:50:16	(270)
fir_8tap_32bit	7:32:04	(66)	10:26:28	(66)	10:46:10	(66)	10:41:42	(66)	9:48:35	(66)	11:45:14	(66)	10:43:40	(66)
fir_pipe_16	23:02:56	(167)	1:00:51:44	(167)	1:00:53:30	(167)	1:00:47:57	(167)	1:00:58:24	(167)	21:40:43	(167)	1:00:38:10	(167)
pipeline_add	3:36:47	(450)	2:46:38	(450)	4:36:46	(450)	5:12:00	(450)	5:21:05	(450)	4:14:22	(450)	3:42:52	(450)
rgb2ycbcr			4:42:04	(163)	6:51:21	(163)	5:51:16	(163)	8:31:02	(163)	7:24:26	(163)	14:28:32	(163)
ter_sum_nine	4:03:30	(114)	5:23:31	(134)	3:10:45	(145)	5:13:45	(145)	3:20:06	(145)	3:55:14	(144)	3:02:24	(145)
weight_calculator	16:21:38	(14)	2:08:03:00	(19)	2:23:49:56	(29)	2:20:44:13	(34)	2:16:30:51	(35)	2:04:18:16	(38)	1:19:16:47	(44)

The runtimes are shown in the format *days:hours:minutes:seconds* and are followed by the average number of performed iterations.
† These benchmarks exceeded the time limit.

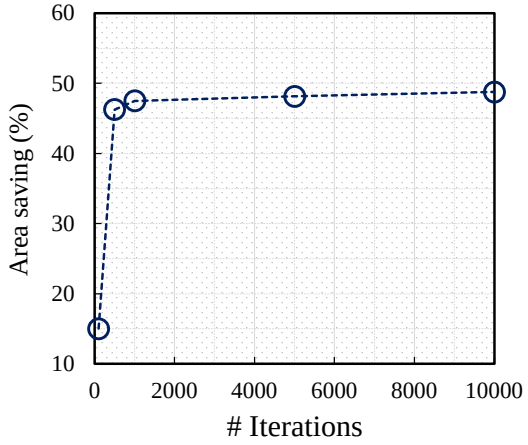


Figure 3.12: Area savings results for FIR filter for different search budgets allocated for MCTS.

would always allow an incremental approximation of the candidates with a step size (e.g., for an adder or multiplier, by truncating/masking one more bit from the least significant side) whereas in case of MCTS-based framework, we selected random approximate transformations to quickly sample wide range of rewards. With random sampling of the approximate transformations, MCTS would be able to exhibit the exploration/exploitation trade-off more efficiently. Finally, selection of candidates for approximation in CIRCA is always in the same order which leads to deterministic results whereas in MCTS-based framework, we allow random selection of a candidate in each depth level of the search tree.

Due to the above-mentioned differences in the underlying setups, a direct and fair comparison of the search techniques in CRICA with the frameworks presented in Section 3.2 was not possible.

To demonstrate the importance of allocating reasonable computational budget to MCTS, we ran the MCTS-based framework with varying number of search budgets for the *FIR filter* and observed how it affects the quality of results which is the hardware area in this case. We represent the results in Figure 3.12 with five different configurations of MCTS with computational budgets of 100, 500, 1000, 5000 and 10000 and are shown as five circles in the graph. Each configuration was run for five times and the hardware area results were averaged. The area savings values for all configurations are shown on the y-axis and are relative to the original area of the *FIR filter*.

It can be observed quite clearly from Figure 3.12 that for extremely lower budgets i.e, 100 iterations, the area savings achieved by MCTS is significantly lower than the other configurations. The performance of MCTS starts to improve from 500 iterations and onwards. However, the improvement trend slows down after 1000 iterations and onwards. We believe that the reason why there is much less difference between the results of 1000 and 10000 iterations is the following: after a certain number of iterations, MCTS typically has collected enough statistics about the tree

nodes to form regions of search where potentially good nodes are present but at the same time it starts to spend most of the upcoming iterations in exploration (finding nodes in unexplored regions) and thus the rate of performance improvement starts to slow down. Nonetheless, we believe that although there are other factors affecting the performance metrics in a search-based optimization process such as the size of the search space, types of approximations that we have in the library, or the sensitivity of the candidates to the error propagation, allocating sufficient search budget is also one of the most important consideration to configure MCTS and must not be overlooked.

3.4 Chapter conclusion

In this chapter, the application of MCTS for the synthesis of *Approximate Accelerator Circuits* (AxACs) is proposed. We show applications of MCTS for synthesis of AxACs via two set of experiments.

In the first part of this chapter, we propose our MCTS-based framework for synthesis of AxACs. The framework is able to approximate design Pareto points by exploring large design space of AxACs. The underlying MCTS algorithm is tailored to suit the AxAC synthesis domain. We note several differences that make AxAC synthesis a different problem than games where MCTS has been primarily applied and gained great success. We then propose a pruning scheme, a novel expansion formula, and a reward formulation for MCTS in AxAC synthesis domain. The pruning scheme helps to cut large portions of unnecessary paths in the search tree. The expansion formula keeps a balance between new nodes added to the tree and the levels / depth of the tree. The reward formulations maps the quality of the intermediate nodes of the search tree to appropriate reward value for the nodes which is later utilized to make selection decisions by the MCTS. The framework is then evaluated with five Verilog-based benchmarks. The results, in principle show that MCTS is a potentially promising method for AxAC synthesis and provide reasonably good results with limited computational budget. It also reveals the efficiency of the proposed framework by resulting in average energy reduction of 28.99% and average area saving of 3.83% than the original circuit for a set of popular benchmark circuits.

In the second part of the chapter, a joint work with the colleagues of our group is presented. This work contributes to CIRCA, an open-source framework, for rapid development and evaluation of different search, approximation and validation techniques for the approximate computing community. Through this work, we integrated MCTS in CIRCA and evaluated its performance. The results of integration of MCTS in CIRCA show that the quality of results greatly depend on the approximation and search methods. Overall, precision scaling performed very well on all the benchmarks by reaching up to 55% area savings. The performance of MCTS lies between hill-climbing and simulated annealing. The main reason why MCTS could

not perform better is too small computational budget allocated to MCTS. Due to the extremely time-consuming verifications that CIRCA performs to validate the intermediate AxACs, MCTS was able to only perform 100 iterations. Since MCTS uses a selection policy based on tree statistics to select nodes in each iteration, it requires sufficiently large number of iterations to be able to make better selection decisions.

Future directions regarding the MCTS algorithm include adding heuristics to the MCTS-based framework to make the search more intelligent. This also could for instance, also employ the error propagation information to predict more suitable approximate operators or nodes in the search tree for subsequent iterations to yield better approximations. Further, the dynamic adaptation of the MCTS parameters such as the expansion parameter (α) and the exploration parameter (C) can be done to change the way the MCTS builds the tree. This could be performed by considering the characteristics of the circuits and designer expectations. To improve the runtime and quality of results for the search, a preprocessing step can be added that could prune unnecessary portion of the search space even prior to start of the actual search.

Chapter 4

Hybrid Methodology for Synthesis of Approximate Accelerators

4.1 Chapter overview

This chapter begins with a motivational example that emphasizes on the effective pruning of the search space for *Approximate Accelerator Circuit* (AxAC) synthesis. Further, it proposes a hybrid synthesis methodology based on *Monte Carlo Tree Search* (MCTS) for the synthesis of AxACs that efficiently handles the problem of *Design Space Exploration* (DSE) as a two-step process. Following that, a comprehensive results section evaluates the proposed methodology. The major contributions are listed below:

- We propose a hybrid technique for the synthesis of AxACs that combines analytical and search-based approximations techniques.
- We adopt a learning-based stochastic search technique i.e., MCTS to sample promising points of the search space based on the statistics of the previous selections. Moreover, we provide a forest of parallel MCTS search trees without any need for synchronization resulting in faster DSE.
- We provide our framework as an open-source contribution and make it flexible and modular to allow its integration with available commercial synthesis tools.

With our proposed hybrid approach, we provide an efficient parallel MCTS implementation based on standard *Message Passing Interface* (MPI) that can leverage any high-performance distributed memory compute facility to speedup the AxAC synthesis process by building up parallel search forest. In our experiments, we extensively utilize the high-performance compute facilities available at *Paderborn Center of Parallel Computing* (PC^2)[1] to run the search-based part of our hybrid AxAC synthesis flow.

[1] `https://pc2.uni-paderborn.de`

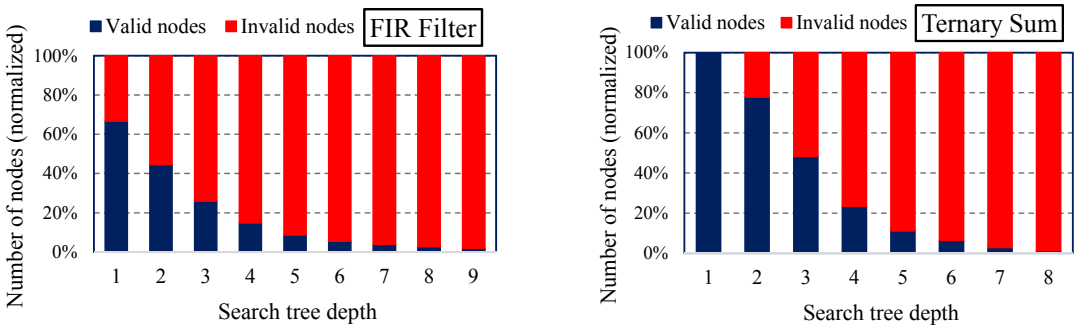


Figure 4.1: Number of valid and invalid nodes in the search space for two example accelerator circuits.

4.2 Motivational example

Any conceivable approximate instance in the design space, called as a node, is considered as a feasible solution if it satisfies the required quality, which is then referred to as a *valid node*. We measure the number of valid nodes for two circuits, FIR filter and Ternary sum, which include 17 and 8 components for approximation respectively, and only 3 approximations for each component. As depicted in Figure 4.1-a and b, the number of valid nodes decreases exponentially as the number of approximations progresses linearly. Typically nodes with more number of approximations lead towards optimal performance parameters but at the same time, to get to such nodes (which lie deeper in the search tree), the search process has to proceed sequentially by validating the nodes through testing/verification which often requires more amount of time [18]. However a great deal of computational time can be saved by directly sampling the desired nodes without invoking time-costly testing/verifications for nodes with smaller number of approximations. In order to achieve this, an efficient DSE methodology is required to prune the non-promising nodes, that encompasses a few number of approximations and explore the promising remaining points toward further enhanced performance parameters.

This idea is further explained in Figure 4.2 with a hypothetical view of the search space growth when approximating an accelerator circuit with nine candidates and two possible approximations for each candidate. The nodes in the search space are all different AxACs. The red nodes are the ones which turned out to be invalid (their error was above the desired quality threshold) after validation whereas the blue nodes are valid nodes (their error was less than the desired error threshold). The gray nodes are part of the search space which have not been checked/validated yet. We also note that the search space shown in Figure 4.2 is divided in to two regions i.e., Region-I and Region-II with Region-I nodes being in unexplored state. The reason why nodes in Region-I are unexplored is that there is a high probability that these nodes will be valid (recall examples of Figure 4.1) so instead of invoking time-costly verification for nodes in Region-I, nodes in Region-II can be sampled and the

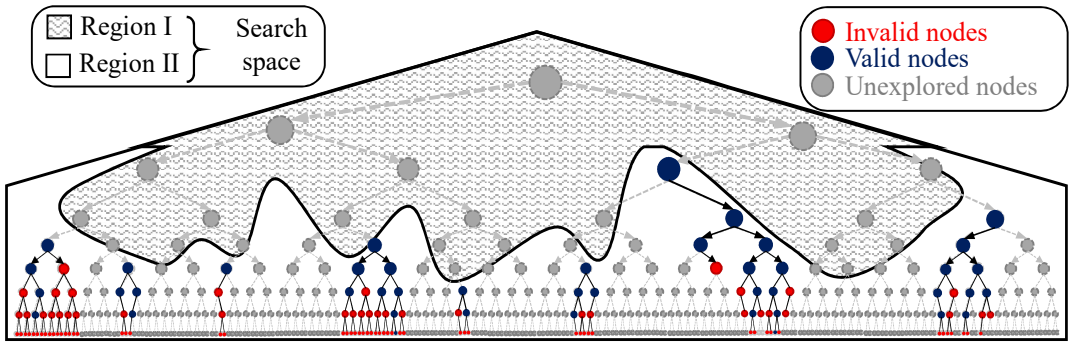


Figure 4.2: Hypothetical design space of a circuit with only 2 possible approximations and 9 candidates. Each node represents an approximate instance. Closer nodes to the root include fewer number of approximations (Region I), while the deeper nodes of the tree have more number of approximations (Region II). Parallel exploration of design space targeting nodes in Region II can considerably improve the runtime of the synthesis process.

search can begin directly from there. Of course, there is no guarantee that a sampled node in Region-II will turn out to be valid and because of the same reason, we suggest to sample multiple nodes in Region-II (as shown in Figure 4.2) and start the search process in parallel from these sampled points. The sampled nodes will be first validated and if they turn out to be valid, the search process uses the node as the root node of the tree. Furthermore, one can eliminate the need for computationally expensive inter-tree synchronization owing to the fact that no two paths in two search trees will be same. Having said that, the questions still remain that how one can sample the nodes and how to determine the border of regions. For the first question, any analytical technique such as precision scaling can be leveraged to randomly generate nodes that have some operators approximated. The answer to the second question varies depending upon various factors such as desired error threshold and the error resilience of the benchmark accelerator circuit. However, one straight forward approach could be to use the operand's effective data range and assign bit-widths accordingly so that the chance of overflow/underflow are minimum. Later, in our proposed methodology, we use an effective statistical technique that provides the effective range with a desired probability of overflow/underflow.

4.3 Proposed hybrid methodology

This section provides an overview of our hybrid methodology which has been illustrated in Figure 4.3. We divide the proposed methodology in two main phases i.e., phase 1 and phase 2 and is preceded by a preprocessing phase. In the preprocessing phase, loop unrolling is performed to uncover further opportunities for approximations. Then sub-circuits in the given input accelerator circuit which are amenable to approximations are identified. These sub-circuits are referred to as candidates. The extracted candidates are then passed on to phase 1.

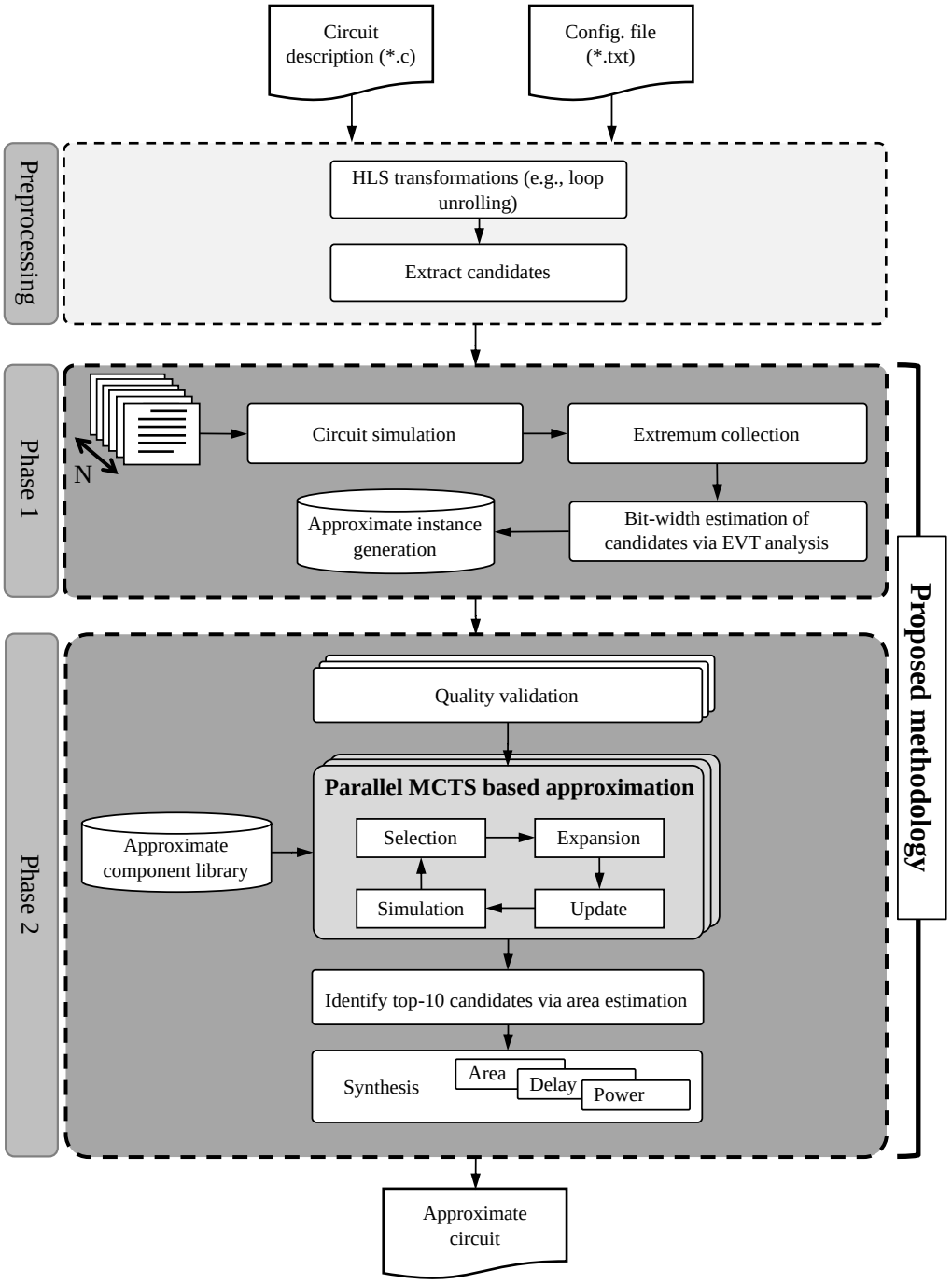


Figure 4.3: Overview of proposed hybrid synthesis methodology.

Phase 1 deals with the analytical approximation of the candidates. It performs circuit simulations using N number of datasets. Each dataset includes input samples generated from a uniform random distribution. The circuit is simulated multiple times with a dataset selected from the available datasets. During each simulation, we record the intermediate values of all candidates and extract the minimum and maximum of these values at the end of each simulation. Using these minimum/-maximum values, we estimate the effective bit-widths of each candidate using an analytical technique explained later in Section 4.4. From the estimated bit-widths, a

population of approximate instances is generated and added to a container used in phase 2 for search-based optimization.

Phase 2 performs a parallel search-based optimization adopted from MCTS [65]. A subset of instances is taken from the container, and each instance is exploited as a root of a search tree. A node in such a search tree represents an AxAC. Since the search trees start at different roots, it is unlikely to explore similar paths during their search and accordingly similar AxACs. Therefore, trees are not required to perform costly synchronization. To build each search tree the root node is validated to make sure that it satisfies the required quality constraint. If it is not valid, the search process is terminated. Then, MCTS starts the search-based optimization and iteratively performs four main operations: *selection*, *expansion*, *simulation*, and *update* until the search budget expires (parallel MCTS based approximation block in Figure 4.3). In the selection step, a node in the existing tree is selected for approximation via a selection policy. The node, which already represents an AxAC, takes one more approximate component from a library and applies it on one of its exact components. The new obtained approximate instance is added as a child of the current node into the search tree. In the next step, the quality of the new node is evaluated through circuit simulation. If the node passes the quality test, it is marked as a valid node otherwise it is marked as dead. Then a reward value is computed by estimating the area improvement archived via the last approximation. For a dead node, the reward is a negative value. This helps the selection step to spend the search budget on the other parts of the search tree. Finally, the reward is back-propagated up to the root node.

When the search budget is exhausted, the built trees are traversed and nodes offering better area savings are selected. The selected nodes are then forwarded for synthesis that reports the area, delay, and power consumption.

4.4 Analytical bit-width estimation through EVT

This section explains the core part of phase 1 of our methodology. First, the underlying extreme value theory is explained followed by its application in bit-width estimation in our approach.

4.4.1 Extreme value theory

Extreme Value Theory (EVT) [78] defines a family of statistical models that captures the behavior of maxima or minima of independently and identically distributed random variables. It is very popular in many areas, e.g., risk assessment, where the behavior of scarce events is entailed to be evaluated. Using a standardized variable $Z = \frac{X-\mu}{\alpha}$, the Generalized EVT distribution is defined by the following *Cumulative*

Distribution Function (CDF):

$$F(Z;\theta) = \begin{cases} exp(-(1+\theta Z)^{\frac{-1}{\theta}}) & \theta \neq 0 \\ exp(-exp(-Z)) & \theta = 0 \end{cases} \tag{4.1}$$

Where μ is the mode of the distribution (sometimes called location parameter), α is the scales parameter, and θ is the shape parameter that controls the type of the EVT distribution. For $\theta = 0$, the distribution is known as Gumbel type, which does not have lower and upper limits and is well suited for identifying the effective range of operands and operations. For $\theta > 0$, and $\theta < 0$ we get the Fréchet and Weibull distributions respectively, which are both bounded on one side.

Parameter estimations and model fitting for a given set of generated extreme points can be performed through several well-known statistical techniques, e.g., *Maximum Likelihood Estimation* (MLE), *Moment Matching* (MM), or *Probability Weighted Moment* (PWM). While MLE and MM can give satisfactory estimations for large samples, PWM provides unbiased and more accurate estimation for a small to moderate number of sample points [79].

4.4.2 Analytical bit-width estimation via EVT

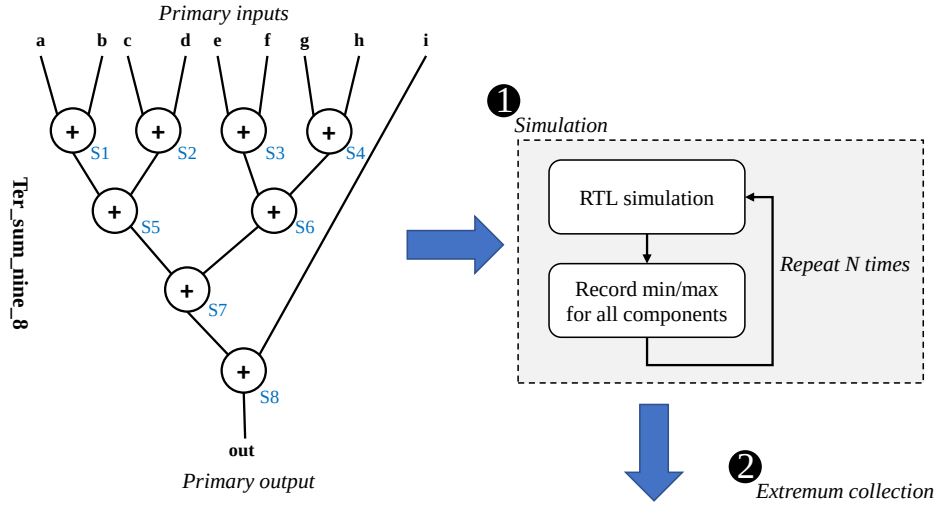
This work exploits EVT as an analytical approximation phase prior to the search-based optimization. In particular, we estimate feasible bit-widths of the candidate operations in an accelerator circuit based on their maxima values via Gumbel type EVT distribution since it does not provide lower and upper limits on the data and thus is well suited for finding the effective data range of operations.

The procedure to estimate the bit-widths starts with the simulation of the target RTL model. For the workload, N independent sets are exploited, in which each set contains input values randomly selected from a uniform distribution. The size of N should be sufficiently large to guarantee that the model parameters are reliable. In this work, N was set to 10^4 with each set containing 10^5 test vectors. The min/max values of each candidate, in the RTL model, are collected and combined to form a so-called extreme value set that later used for EVT CDF parameter estimation.

The process is further elaborated in Figure 4.4 for the *Ter_sum_nine_8* circuit, one of the benchmarks that we used to evaluate our methodology. During the simulation step that is repeated for N iterations, randomly selected test sets are applied to the inputs of *Ter_sum_nine_8*. The primary output as well as the intermediate outputs of the components (in this case adders marked as $S1$ to $S8$) are recorded for each test set. After all the test vectors have been applied, the minimum and maximum value of each component for the current input test set is extracted.

After N iterations, we collect N minimum/maximum values for each component in a table as shown in Figure 4.4.

Next, we need to find a threshold value τ_{max} that all possible maxima values taken by a component (denoted as X) are less or equal to τ_{max} with the probability



N	S1		S2		S3		S4		S5		S6		S7		S8	
	Min.	Max.	Min.	Max.	Min.	Max.	Min.	Max.	Min.	Max.	Min.	Max.	Min.	Max.	Min.	Max.
1	9	505	11	495	11	498	23	507	100	935	73	941	292	1659	354	1817
2	11	507	9	507	6	507	15	505	105	911	115	936	382	1646	459	1824
3	6	497	14	503	4	507	8	493	122	986	69	957	373	1687	535	1811
4	15	497	4	501	3	492	13	498	76	915	44	939	297	1755	452	1945
…																
10000	1	501	22	499	4	491	9	509	71	924	91	944	388	1687	445	1761

Figure 4.4: An example showing how the extreme values are collected for *Ternary sum* benchmark.

of ξ as stated in the following equation:

$$F(Z;\theta) = P(\frac{X-\mu}{\alpha} \leq \tau_{max}) = \xi, \qquad 0 < \xi < 1 \tag{4.2}$$

Here ξ states how much of the sample points will be less or equal to τ_{max}. For instance, if we have $\xi = 0.95$, this means that the 95% of the sampled maxima values are expected to be smaller than τ_{max}. This is graphically illustrated in Figure 4.5.

Next, we observe that since the EVT provides an unbounded estimator (beyond τ_{max}), finding the optimum bit-width of available candidates using EVT estimation imposes a considerable cost of testing/verification [80]. In fact the bit-width estimate

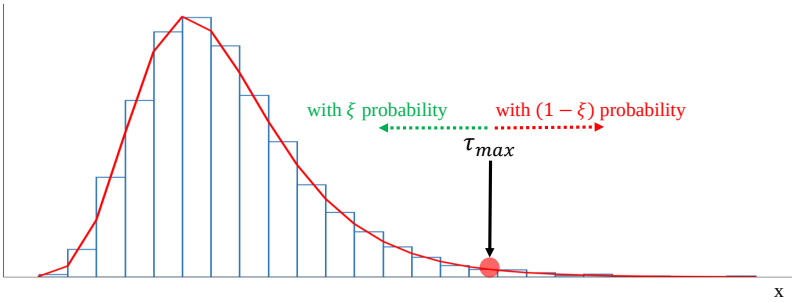


Figure 4.5: Gumbel distribution.

obtained through EVT might cause overflow since it is only utilized as an estimate or in other words as an approximation that might sometimes results in error at the output of the operations with a probability of $1 - \xi$. Keeping this in mind, we generate sufficient configurations of approximate instances having different bit-widths sampled from the range provided by the EVT, so that later in the search-based optimization phase, enough sample points are available to start exploring the design space. Later on, we only need just one testing/verification prior to the start of the search to decide whether to continue the search from that point.

The most probable range of each candidate is determined through selected boundary thresholds (τ_{min} and τ_{max}) and the estimated bit-width is computed as follows:

$$estimated_{bit-width} = \begin{cases} \lceil \log_2(\mid \tau_{min} \mid) + 1 \rceil & \tau_{min} < 0 \\ \lceil \log_2(\tau_{max} + 1) \rceil & \tau_{min} > 0 \end{cases} \tag{4.3}$$

Where the $\mid \tau_{min} \mid +1$ states that the estimated value requires one more bit of sign.

4.5 Parallel search-based optimization

The phase 2 of our hybrid methodology is based on the adaptation of the MCTS. Since the details of the basic MCTS algorithm have been explained already in Chapter 3, we only explain here the modifications and our parallel implementation of the MCTS algorithm.

4.5.1 Parallel MCTS

MCTS is a powerful algorithm especially for problems with a large search space since it modifies the search strategy based on the statistics of already explored nodes and intelligently prunes the non-promising nodes as the search progresses. However originally suited for games domain, the MCTS algorithm when applied to the domain of AxAC synthesis, needs adaptations owing to various characteristics of hardware synthesis flow. We explained most of the adaptations in detail in Section 3.2. Remaining are highlighted in the following as they are discussed in our proposed DSE algorithm.

Algorithm 3 represents the major steps of the proposed parallel search-based optimization using MCTS. The algorithm takes a subset of already approximated instances, generated in phase 1, from the container and forms a forest of parallel search trees, in which each instance is the root of a search tree. The number of parallel trees is determined by the designer with a parameter i.e., k and the search budget M determines the number of iterations for the search trees and can also be set along with many other parameters using a configuration text file.

In the parallel search-based optimization, we use a master-slave process model to handle the search. The master process is responsible to initiate root nodes and invoke other processes that ultimately grow the search tree. The master process

Algorithm 3: Parallel search-based optimization

```
   Input: O= Original accelerator circuit(*.c), k = number of search trees,
   M = search budget
   Output: A={Approximated instances satisfying quality constraints}
 1 S ← select a subset from the container
 2 do in parallel
 3     root_node ← get root_node from S
 4     valid ← validate the root_node
 5     if valid then
 6         while M > 0 do
 7             curr_node ← select via UCT                    // Selection
 8             if curr_node is not leaf node then            // Expansion
 9                 node_new ← a new child node of curr_node
10             else
11                 continue
12             valid ← simulate(new_noode)                   // Simulation
13             if !valid then
14                 Mark new_node as dead
15             reward ← compute_reward(new_node)
16             update(reward, new_node)                      // Update
17             M ← M − 1
18     else
19         take another node from the container and repeat the algorithm from 2
20 end
```

initially selects a subset of root nodes samples in Phase 1 and populates a list S which it later distributes to the other processes (Line 1 in Algorithm 3). Following that, each process gets a root node from S and start building their individual search tree in parallel (Line 2 to Line 19). Each process performs a quality validation of their root node prior to expand the search tree to be sure that they do not violate the quality constraint (Line 4). In case of violation, another instance is taken from the container until we have k number of valid root nodes (Line 19).

Then the search trees iterate over the selection, expansion, simulation, and update steps in parallel till the simulation budget of each tree, i.e., M, is exhausted. For selection of a node in the search tree, we use a formulation similar to the UCT with a slight modification. Instead of starting at root node and iteratively selecting child nodes based on their UCT scores, we sort *all* the active nodes in the tree built so-far based on their UCT score and then select the node with the highest UCT score. The rationale behind this selection is that the nodes deeper in the search will have higher average rewards (since their visit count is low) if they turn out to be valid and in fact they should be preferred than the nodes near the root node. Our experimental results confirm that this assumption is correct.

Table 4.1: Benchmark accelerator circuits

Circuit	I/O	Candidates	Trans.†	QoR §	Area (μ^2)*	Power (mW)*
Convolution filter	8/8	17	19	PSNR	31 150.44	0.72
FIR filter	16/32	17	21	MRE(%)	29 829.94	13.50
RGB2YCBCR	8/8	17	21	MRE(%)	30 337.18	3.40
Ter_sum_nine_8	8/16	8	10	MRE(%)	950.58	2.16
Ter_sum_nine_16	16/32	8	10	MRE(%)	1007.98	2.16

† Approximate transformations from the EvoApproxLib library [46].
§ Error metric used for quality of results.
* The area and power of the accelerator circuits were measured using Synopsys Design compiler using a 22nm technology library

The expansion step adds a new node to the tree by applying one more approximation on the selected node (Line 9). In case the selected node can not be further expanded, the current iteration is completed and the algorithm starts a new iteration. The simulation step (Line 12) checks whether the node satisfies the quality constraint and if found invalid, the node is marked as dead node. The subsequent step computes a reward for the new node. We use circuit area as the performance metric for optimization. It is obvious that other metrics such as power consumption and delay can be utilized as well. For every node, we compute a reward based on its area saving. To find the area improvement statistics, the synthesis process would be very costly. Instead, we provide an area saving estimation using pre-computed area values of the components in the approximate library as follows:

$$R(new_{node}) = \frac{\mid Area_{AxC} - Area_{Ori} \mid}{Area_{Ori}} \tag{4.4}$$

Where $Area_{AxC}$ refers to the area value of the approximation component exploited in the current node and $Area_{Ori}$ refers to the area of that component without approximation.

For invalid nodes however, the reward is a negative value representing a penalty for the selected node. The update step then propagates the reward all the way back up to the root node for all nodes on the selected path. After the allocated search budget expires, the valid nodes explored by all the processes are analyzed and top instances are selected and synthesized.

4.6 Experimental results

4.6.1 Setup of experiments

The benchmark accelerator circuits that we used to evaluate our methodology are shown in Table 4.1. All of the benchmarks were coded in *SystemC* since it facilitates rapid high-level design prototyping. In the preprocessing step, all loops were flattened and candidates were extracted from the original circuit using the annotation.

As the error metric, we use PSNR for *Convolution filter*. For all other benchmarks, we use *Mean Relative Error Percent* (MRE(%)) to evaluate the quality.

We implemented the bit-width estimation phase in Python using *scikit learn* library and the search-based optimization phase in C++ via *Openmpi v3.1.3* for parallel execution of multiple search trees. The analytical part of our experiments was performed on a compute cluster running a scientific Linux 7.2 (Nitrogen), comprising of 16 nodes with an Intel® Xeon E5-2670 @ 2.6GHz and 256 Gigabytes of main memory, of which it provides 2 Gigabytes per job. For the first phase, we use $N = 10000$ and each dataset contains 10^5 random samples from a uniform distribution. We also chose k = 20 as the number of parallel search trees. The generated approximate designs were functionally validated with datasets comprising of 10^5 samples generated with a uniform distribution for each benchmark, except the *Convolution filter* for which we used 500 samples from the test set of CIFAR-10 [81]. The candidates in Table 4.1 represent only the data-path elements and we do not approximate parts of circuit contributing to control-path. We used approximate transformations from an open-source approximate components library EvoApproxLib [46]. Approximate designs were synthesized using *Synopsys Design Compiler version K-2015.06* (Synopsys-DC) with a 22nm target technology library. To evaluate power consumption, we first obtained the *Switching Activity Interchange File* (SAIF) of each benchmark via simulation using their corresponding datasets and then injected the SAIFs to Synopsys-DC for measurement.

Our parallel implementation of MCTS exploits *Message Passing Interface* (MPI) to distribute the workloads among the compute nodes. In our implementation, we used a trivial scenario typically referred to as master-slave in context of MPI. To run our MPI implementation, we utilized the compute cluster facility of *Paderborn Center of Parallel Computing* (PC^2), namely OCuLUS which consists of 552 small and 20 large compute nodes each with two Intel® Xeon E5-2670 processors running at 2.6 GHz with 64 and 256 GByte of main memory, respectively.

The flow of our MPI implementation is shown in Figure 4.6. Here, the *mpirun* command initiates our MPI application to 20 compute nodes hereinafter referred to as *processes* numbered from 0 to 19 in this case. All processes perform initialization in parallel as shown in step 1 in Figure 4.6 after which the execution flow for process 0 (master) and process 1-19 proceeds differently in parallel. Process 0 enters in to a loop where it continuously assigns jobs (search tree seeds in this case), and receives results until all the search trees have been completely distributed (step 2, 3, and 4) and finally terminates the processes by sending a die signal (step 5). The slave processes wait for job allocation from process 0 and start building the search trees independently once they receive job from process 0. Once they consumed the search budget, they send back their results to process 0 and then wait for signal from the process 0. Once receiving a die signal, the processes terminate.

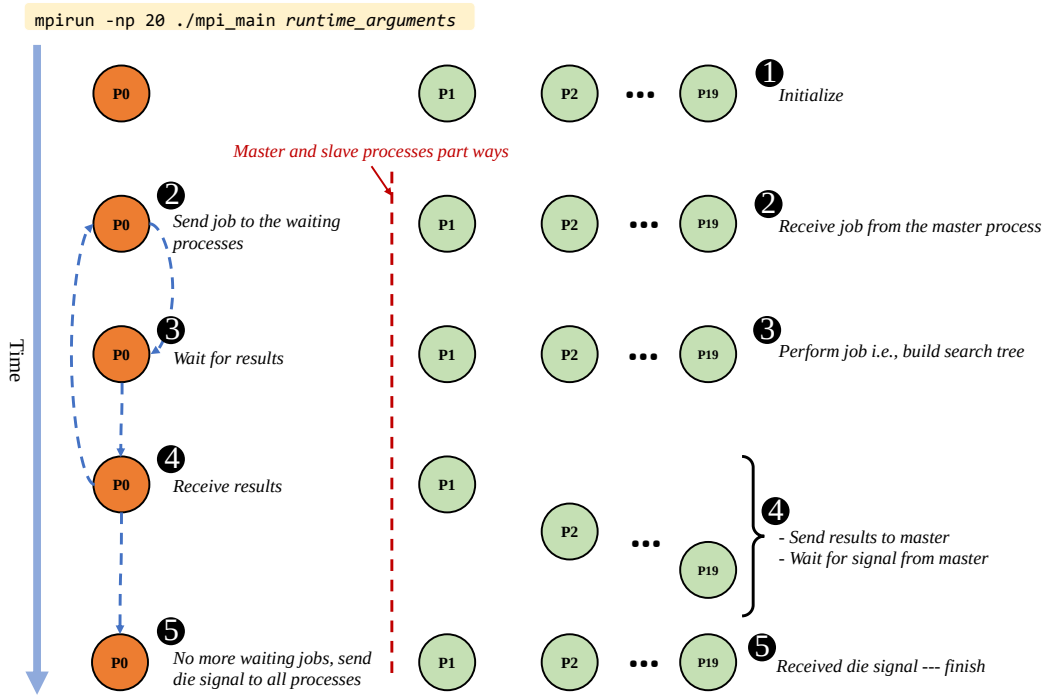


Figure 4.6: Flow of the MPI implementation for parallel MCTS.

4.6.2 Results and discussion

To evaluate our methodology, we performed experiments for a gird of values for maximum error bounds of benchmarks. Beside our hybrid methodology (HM), we implemented two other techniques as the baseline for comparison: 1) MCTS: this is a single tree MCTS search-based approach and 2) Min_BW: this is the minimum bit-width instance of the benchmarks. To perform a fair comparison, the search budget of the single tree MCTS is selected as the sum of the simulation budget of search trees in HM. The Min_BW is realized via iterative bit-width optimization using the method we introduced in phase 1.

Figure 4.7 shows the results for area savings of HM and MCTS for 10 configurations of error bounds for the given 5 benchmarks. The HM outperforms MCTS in all the configurations for all the benchmarks. In particular, HM was able to achieve a maximum of 10.57× area saving against MCTS for *Convolution filter*. The minimum saving was in case of *RGB2YCBCR* where it can still achieve 1.02× saving than MCTS.

Figure 4.8 highlights the power saving values. Even though the HM targets area as the main performance metric, it can be depicted from Figure 4.8 that HM is also capable to reduce the power consumption for most of the error bound values in the given benchmarks as well. Again, for the *Convolution filter*, HM achieves maximum power saving of 2.7× to that of MCTS. For one error bound (1.5%) of *FIR filter* and three error bounds (0.5%, 3.0%, and 5.0%) of *Ter_sum_nine_8*, the power consumption values obtained by HM are up to 4% higher than MCTS. In all other cases, HM provides at least the same or less power consumption values.

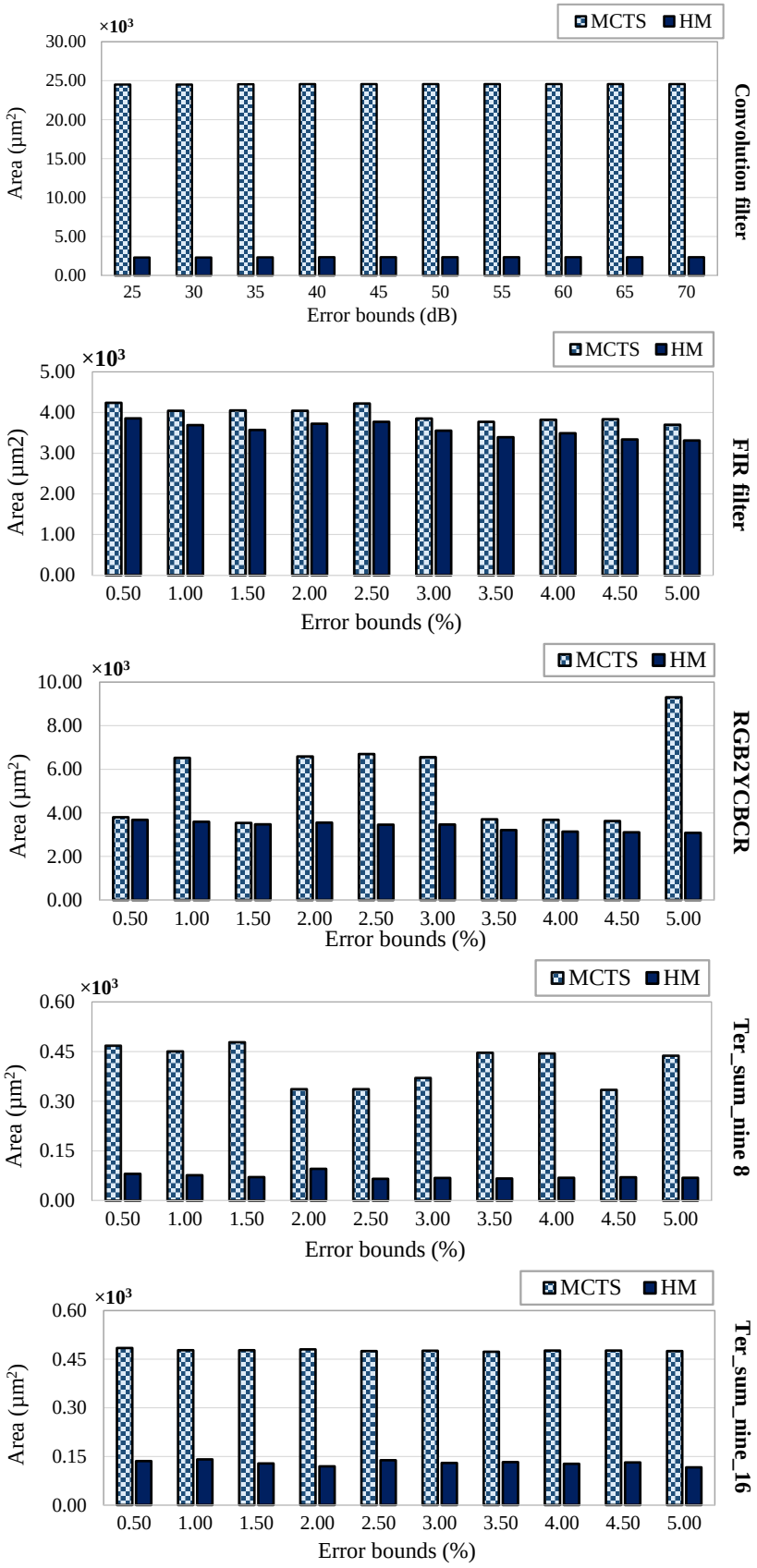


Figure 4.7: Area savings of HM against MCTS for five different benchmarks on various error bounds.

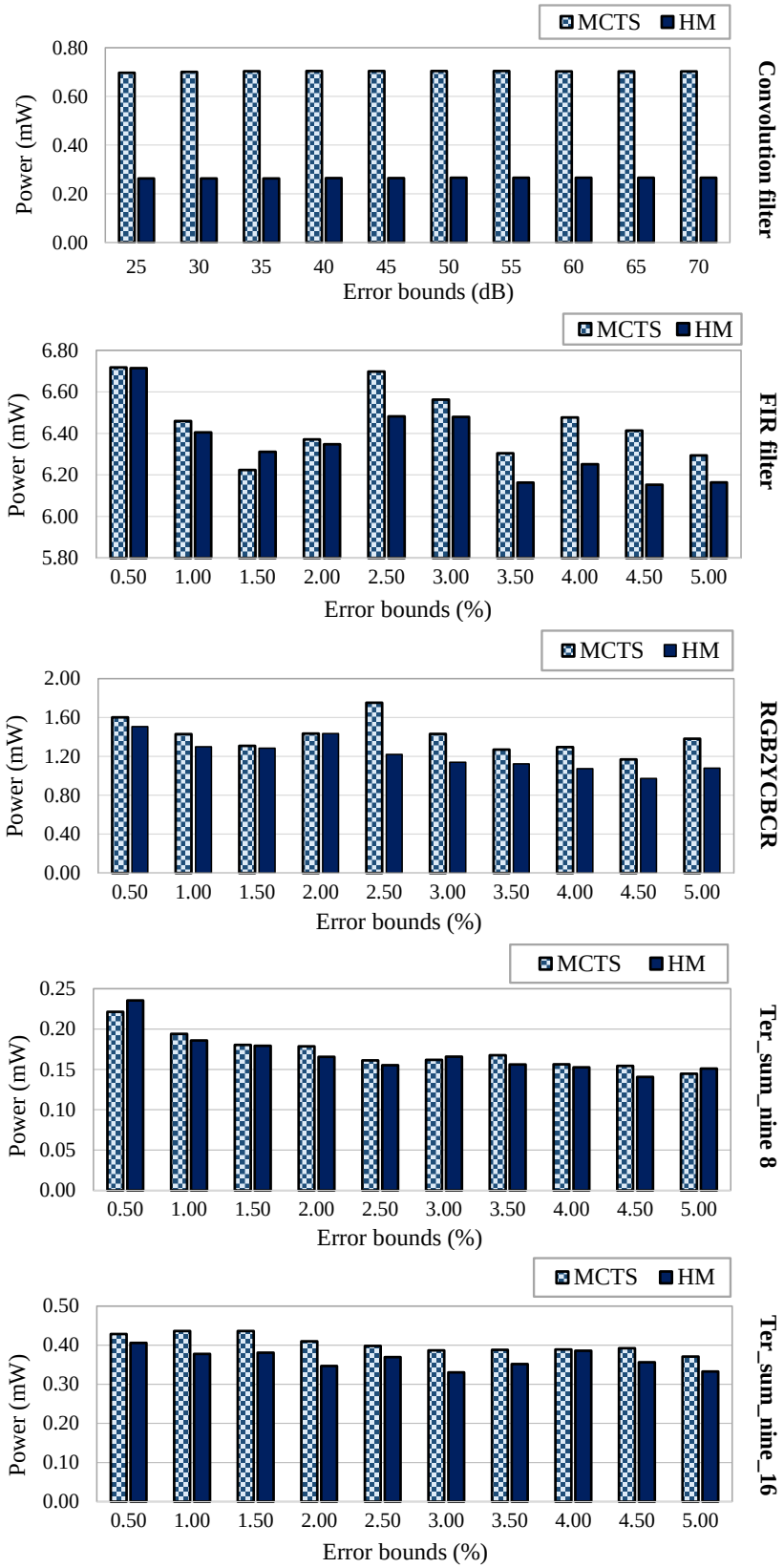


Figure 4.8: Power savings of HM against MCTS for five benchmarks.

In Figure 4.9 and Figure 4.10, the normalized area and power savings of HM and Min_BW are illustrated, respectively. For HM, we report the maximum savings

obtained among all error bound values. Again, HM demonstrates its effectiveness against Min_BW by reaching up to 39.19% more area saving for *Ter_sum_nine*_16 and 54.63% more power saving for *Ter_sum_nine*_8. The minimum area and power savings of HM are 4% and 1% both for *Convolution filter*, respectively.

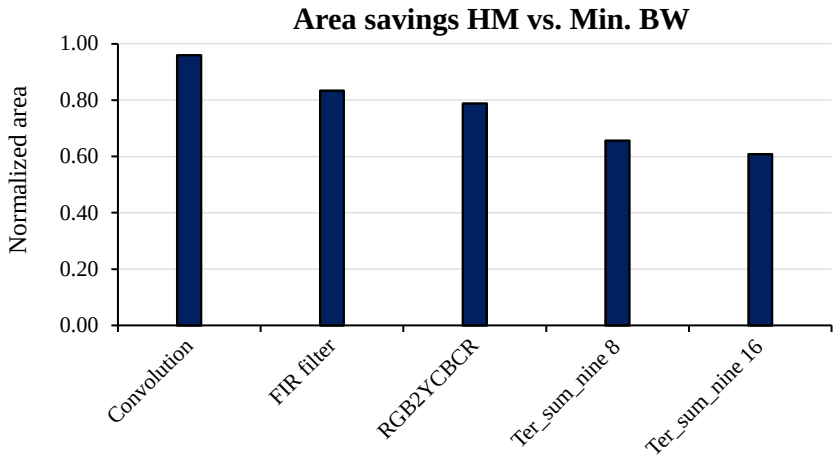


Figure 4.9: Area savings of hybrid approach against purely analytical approach.

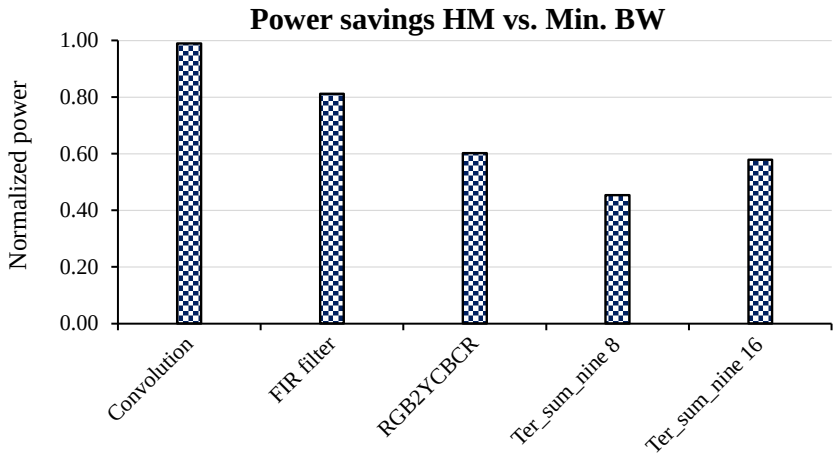


Figure 4.10: Power savings of hybrid approach against purely analytical approach.

The runtime for HM and MCTS are shown in Table 4.2. Although the HM and MCTS approaches use the same computational budget, HM is able to complete the exploration in less time since it does not spend much time like MCTS to evaluate nodes with just a few number of approximations. HM starts the search in depth of the design space where there are lots of dead nodes, and moreover benefits from the parallelism, hence it completes the search faster. In particular, HM reaches up to 16.56× speedup against MCTS for *Ter_sum_nine_8*. The minimum speedup that HM achieves against MCTS remains 2.77× for *Convolution filter*.

4.7 Chapter conclusion

In this chapter, we have presented a hybrid synthesis approach for *Approximate Accelerator Circuits* (AxACs) that exploits both analytical bit-width estimation as well as search-based optimization to quickly find feasible AxACs that satisfy the required quality. Our approach quickly reaches to the interesting areas of the search space without performing costly validations via analytical bit-width estimation phase and then fine-tunes the approximations by performing stochastic parallel search in which the approximate transformations from an open-source library of approximate components are applied to get more area savings. We show with experimental results that the hybrid approach is capable to achieve substantial area savings than both purely analytical and purely search-based methods. Moreover, with clear separation of both phases, the proposed approach can allow any analytical and search-based optimization method to be used in combination to produce AxACs. To this end, we provide our proposed methodology as an open-source contribution to the community[2].

In future, more analytical approaches will be evaluated for first phase of the proposed hybrid methodology. Furthermore, parallel implementation of the MCTS forest can be improved by introducing light-weight inter-tree communication to share information about the approximate transformations and candidates. For instance, if a particular transformation turns out to be significantly better than other transformations, it can be communicated at a global level to let other search tree use this information. This can potentially improve the overall quality of results.

[2] `https://git.upb.de/ceg_upb/hybrid_axc_synthesis`

Table 4.2: Runtimes for MCTS and HM approaches for various error bounds

	ERROR BOUNDS									
BENCHMARKS	**0.5%**	**1.0%**	**1.5%**	**2.0%**	**2.5%**	**3.0%**	**3.5%**	**4.0%**	**4.5%**	**5.0%**
					MCTS					
FIR filter	2:09:58:59	1:22:53:14	1:14:51:32	23:28:05	1:17:17:05	19:34:02	20:21:26	20:41:56	20:12:38	1:20:46:41
RGB2YCBCR	2:11:56:47	2:00:16:37	1:16:31:18	2:14:06:57	1:18:51:49	21:21:43	22:15:51	2:11:56:59	21:16:33	21:24:13
Ter_sum_nine_8	23:34:44	19:35:24	2:08:36:30	2:09:21:45	18:46:48	18:47:58	2:07:02:19	17:26:53	19:12:59	1:19:18:38
Ter_sum_nine_16	19:48:03	2:06:30:23	2:06:28:11	2:06:52:23	18:34:28	18:12:23	18:14:26	17:21:13	17:58:32	17:50:29
	25 dB	**30 dB**	**35 dB**	**40 dB**	**45 dB**	**50 dB**	**55 dB**	**60 dB**	**65 dB**	**70 dB**
Convolution filter	3:03:35:29	1:07:05:18	2:06:54:18	2:09:27:41	1:23:48:22	1:17:18:24	2:09:48:34	2:12:47:52	1:22:57:32	1:17:17:10
	0.5%	**1.0%**	**1.5%**	**2.0%**	**2.5%**	**3.0%**	**3.5%**	**4.0%**	**4.5%**	**5.0%**
					HM					
FIR filter	10:04:03	10:39:44	18:15:14	11:12:07	10:46:05	12:25:22	10:47:00	9:51:19	10:51:49	10:47:25
RGB2YCBCR	10:37:33	11:37:17	15:16:48	18:01:51	10:35:49	11:28:39	12:10:40	10:38:10	10:35:04	11:36:29
Ter_sum_nine_8	54:51	1:28:41	1:24:28	1:21:32	1:26:42	3:07:17	2:10:07	2:33:34	2:16:23	2:50:38
Ter_sum_nine_16	1:47:23	1:07:24	2:13:24	2:15:11	1:32:37	3:46:47	2:01:45	2:37:24	2:39:10	3:35:20
	25 dB	**30 dB**	**35 dB**	**40 dB**	**45 dB**	**50 dB**	**55 dB**	**60 dB**	**65 dB**	**70 dB**
Convolution filter	21:45:14	0:21:30:32	16:03:25	13:13:31	11:36:21	13:07:25	11:04:35	10:30:48	9:28:12	10:16:13

All runtimes are in *days:hours:minutes:seconds* format.

Chapter 5

Machine Learning-based MCTS

5.1 Chapter overview

In this chapter, an improved MCTS-based *Approximate Accelerator Circuit* (AxAC) synthesis flow based on *Deep Neural Network* (DNN) error estimation models is presented. The proposed approach avoids time-costly simulations / validations by relying on trained DNN models to obtain the error information and thus provides tremendous speedup as compared to the traditional simulation based flow.

The first part of this chapter explains the setup and training of DNN error estimation models. In the later part, experimental results are provided that demonstrate drastic improvements in the runtime over a simulation-based greedy and MCTS AxAC synthesis baselines. The contributions of this chapter are highlighted in the following.

- We develop high-accuracy fast DNN-based error estimation models to learn the error propagation of operators due to approximation in a number of practical application benchmarks.
- We demonstrate the use of DNN-based error estimation models in a search-based AxAC synthesis flow driven by MCTS and achieve similar or even better quality of results for hardware area with considerable reduction in the runtime.

5.2 Motivation and background

As discussed in the Chapter 1, the design space for AxAC synthesis grows exponentially with the increasing number of operators and approximate transformations. This not only results in very long runtimes but also puts restriction on the number of iterations performed by the search-based process and hence large part of the design space remains unexplored.

Majority of existing frameworks handle the *Design Space Exploration* (DSE) with a search-based approach [15, 16, 51, 57, 18, 22, 23]. The search-based flow generally consists of the following main steps: (1) expansion of search space by creating new approximate instance with incremental approximation on one of the already

added instances and (2) validation of newly generated design instance by evaluating the error and impact on the target metric for the generated instance. For (1), the search method's policy determines which instances to be selected and expanded whereas for (2), the validation is mostly achieved via formal verification or testing-based methods.

Although formal verification-based methods can provide strong guarantee on the worst-case error of an approximate instance, the runtimes are relatively very long compared to a testing-based approach which evaluates the error by application of a reasonably large subset of input vectors to the design under test [18]. Even with a testing-based approach, the validation times are relatively very high as compared to the other steps (of search) and dominate the total time spent in the DSE. Due to this bottleneck, the existing search-based DSE frameworks need to limit the number of performed iterations leaving a large portion of design space unexplored. Many existing frameworks have employed heuristic search methods (such as greedy) to truncate large number of intermediate designs in an effort to reduce the size of the design space during the process [15, 22, 18, 14]. This however results in suboptimal outcome due to discarding many search regions with potentially better solution quality.

In Chapter 3, we presented an MCTS-based AxAC synthesis framework based on a balanced tree selection policy allowing it to explore design space with a mix of exploration and exploitation. In this way, the DSE phase had a lower chance of getting stuck in a local minimum. Later in Chapter 4, the MCTS-based DSE was combined with a preceding analytical phase that helps skipping a large number of validations by sampling multiple design points in the search space. These design points then act as root nodes for multiple search trees that are then explored in parallel. Although being able to explore the search space more efficiently and eliminating large number of unnecessary validations, still the verification time remains a bottleneck for such approaches putting a limit on the number of iterations.

In general, the mentioned limitations of existing search-based frameworks put them at disadvantage for two aspects. These frameworks either spend too much time for too few iterations and are not able to explore large part of the design space (e.g., MCTS) or they discard a substantial portion of the design space that could potentially offer better quality of results (e.g., greedy). Nevertheless, both issues are linked with the validation bottleneck which consumes most of the time in the DSE. In this chapter, a faster DSE approach is proposed that relies on accurate and fast error estimation models obtained via DNNs to drastically reduce the runtimes of the search-based DSE methods.

5.3 DNN enabled MCTS-based AxAC synthesis framework

This chapter proposes *Machine Learning-based MCTS* (ML-MCTS) for AxAC synthesis. The objective of ML-MCTS is to improve the runtime efficiency of the MCTS-based AxAC synthesis framework via DNN-based error estimation. This section deals with the explanation of the overall flow and the underlying key phases of the ML-MCTS framework.

The overall flow of the proposed improved framework is depicted in Figure 5.1. ML-MCTS contains two main phases together with a preprocessing and a post-processing step. The preprocessing phase performs two jobs. First, it looks for loops in the source code and if found, unrolls / flattens them. This uncovers more room for approximations. Second, it extracts the candidates from the source code.

The first main phase of ML-MCTS is called regressor training phase. This phase accepts the extracted candidates and the original source code from the preprocessing phase and prepares an RTL simulation environment. During the RTL simulation,

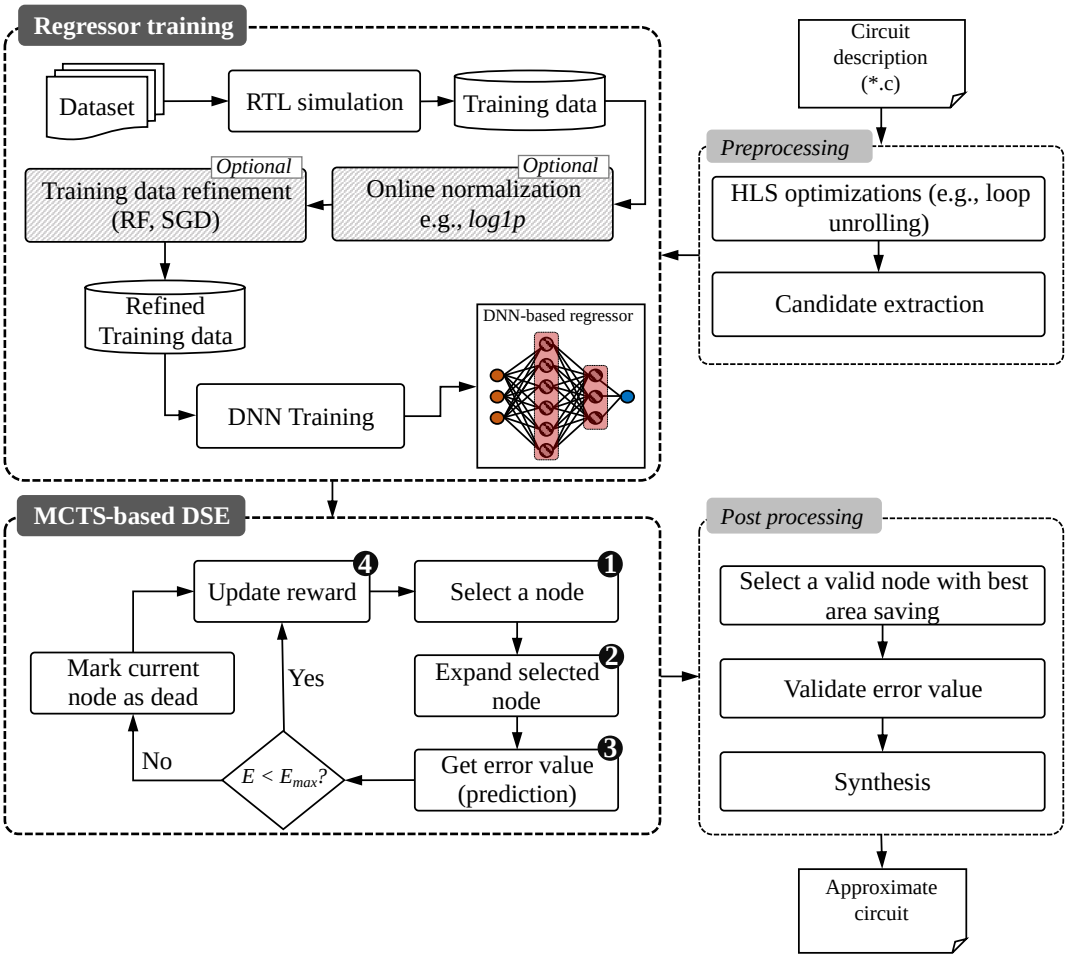


Figure 5.1: Overall flow of the ML-MCTS framework.

AxACs are repeatedly generated with different configurations with varying approximations degree i.e., number of candidates approximated out of total available candidates. Each AxAC instance generated during the simulation phase is evaluated with a set of random input samples to evaluate the error caused by approximation. Then, the AxAC's configurations along with the observed error forms a data sample which is then added as one row of the training dataset. The RTL simulation process is repeated for predefined number of iterations and results in a training dataset (as shown in Figure 5.1). The generation of training dataset is further explained in Section 5.3.1. The collected training dataset from the simulation step is then refined if required through two optional steps. A normalization step for instance might be required if the range of the observed error does not fully capture the behavior of the error. Moreover, we employ simple yet powerful classification models such as those based on random forests to further refine the dataset prior to train the DNN (Training data refinement step in Figure 5.1). Finally, training of the neural network is performed using the refined training data and the resultant model is saved for the next phase.

The second main phase relies on a search-based optimization to find the best approximate design in terms of the target metric (in this case the hardware area) while adhering to the given error bounds. This is achieved via *Monte Carlo tree search* (MCTS), an intelligent stochastic search technique specially suited to problems with large branching factor. MCTS performs four main steps iteratively to generate a search tree where the nodes of the tree represent different AxACs. This phase is explained in detail in Section 5.3.2. However, it is important to mention here that the key to reduce the runtime of the DSE phase in the proposed ML-MCTS framework is the simulation step which invokes fast DNN-based inference engine that was trained in the previous phase to obtain the circuit error information. This enables extremely fast quality validation thus reducing the over all runtime drastically as compared to a testing-based simulation phase. The DSE phase iteratively grows the search tree until the allocated search budget expires. The result of this phase is the list of active nodes in the search tree.

The active nodes explored by MCTS phase are then analyzed in a post-processing step to find a node with best area savings. The node is then synthesized and the area and power values are reported.

In the following subsections, the two main phases of the ML-MCTS framework are explained in detail.

5.3.1 Deep neural network based error estimation regressor models

The regressor training phase of ML-MCTS starts by configuring an RTL simulation environment. During an iterative process, it generates approximate variants of the original circuit with different configurations of approximations. This iterative process is illustrated in Figure 5.2 where as an example, we show how the training data for *RGB2GRAY* benchmark accelerator circuit (one of the benchmarks which we later

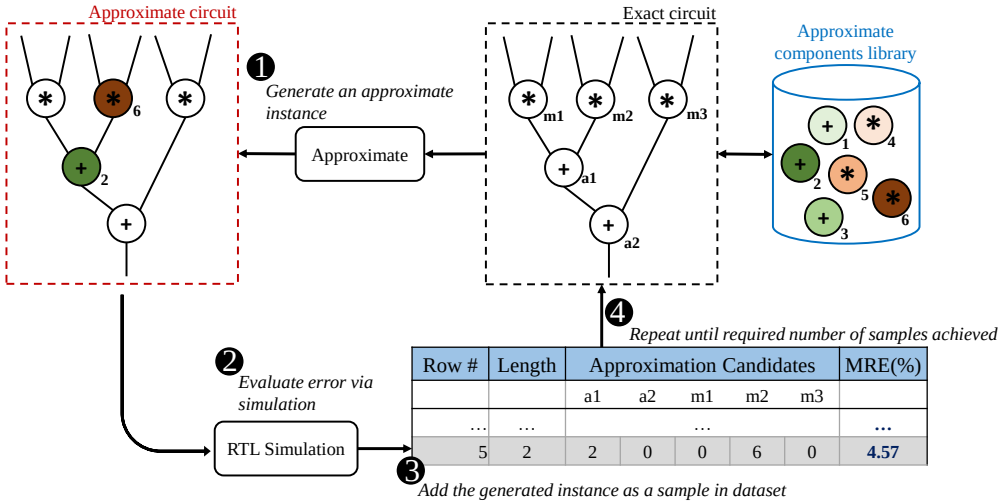


Figure 5.2: Example showing how the DNN training data is formed.

use to evaluate the proposed framework) is collected. The benchmark circuit is composed of three multipliers (referred to as *m1*,*m2* and *m3*) and two adders (referred to as *a1* and *a2*) in the exact circuit shown in Figure 5.2. An approximate component library is available and contains approximate versions of both multiplier and adder components. In this example, the library provides three approximate versions of each component with varying characteristics of error and target metrics improvements. The approximate components are enumerated with unique identifier i.e., 1 to 6. For approximate components, we use a subset of 16-bit adders and multipliers from EvoApproxLib which is an open-source library of approximate components generated via Cartesian genetic programming approach [46].

In step 1 an approximate version of the original accelerator circuit is obtained by replacing some (or all) components with one of the available approximate components from the library. In this work, we randomly select the number of candidates to be approximated in each iteration. In the example shown in Figure 5.2, two components i.e., the adder *a1* and the multiplier *m2* is selected for the approximation. We refer the number of components approximated as length of approximation and use it as one of the features to be fed to our neural network. The generated approximate instance is then simulated using a test bench. During the simulation, test vectors generated with uniform random distribution are applied to the circuit under test. The size of test vectors is kept sufficiently large enough to capture the output error behavior. We use one million test vectors for signal processing and arithmetic benchmarks whereas for image processing benchmarks, we use 500 samples from the test set of CIFAR [81].

The RTL simulation reveals the quality of the approximate instance (which is in this case 4.57%). Then, in step 3, we form one sample of the dataset from the AxAC's configuration, the length, and the observed error as shown in the table in Figure 5.2. Here, the first column represents the row number of the data set and goes from 1 to 100,000. The second column (Length) provides the number of approximations

applied in that configuration row or in other words, how many candidates (out of total) are approximated in that row. Following that, there is a column dedicated to each candidate (in this case *a1,a2,m1,m2,m3*) and the value in the corresponding cell either contains a zero which means the candidate is not approximated or it has a number representing that it is approximated and the number in that case represents the unique identifier of the approximate component. The last column gives the error measured with the simulation. It should be noted that with this scheme, the topology of training data for all benchmarks remains the same and we do not need to incorporate this data in the configuration file.

The process continues to generate approximate configurations and updating the dataset until the required number of samples have been obtained. For each benchmark, we generated 100,000 data rows to constitute a dataset (referred as training data in Figure 5.1).

However, we observe that the training data is not always ready to be directly used for training purpose because of irregularities in the distribution of the error. For instance, in case of the image processing benchmark i.e., *Gaussian blur*, we observed that the error distribution has two separate and distant regions i.e., one representing the error distribution for imprecise designs (e.g., $PSNR$ in range of $0 - 61$) whereas the other case of error showing a very large value of $PSNR$ (i.e., > 99 including ∞). This is reflected by the far right bar in the histogram of Figure 5.3).

For such cases, we adopted the strategy of a two-layer error prediction: the input is first presented to a classifier which provides a binary output showing whether the design represents a precise case ($PSNR > 99$ or ∞), in which case the input design configuration is considered as precise. If the classifier predicts otherwise, a trained regressor model is invoked which predicts the error magnitude (in this case the $PSNR$).

We evaluated different well-known classifier models and obtained significant accuracy with the random forest [82] and stochastic gradient descent classifiers [83].

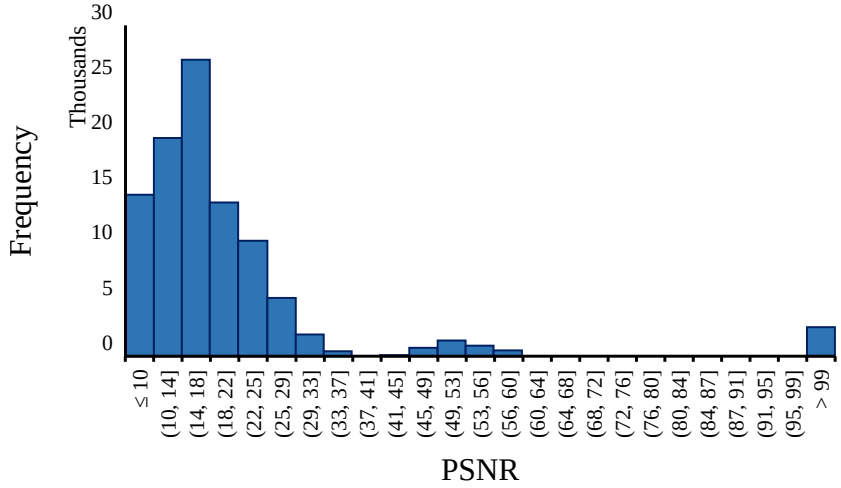


Figure 5.3: Error distribution for Gauss_blur benchmark.

Table 5.1: Regressor training parameters

Benchmark	Features	Prediction setup	R^2
FIR filter	18	Regressor	97.52%
RGB2GRAY	6	Regressor	72.35%
Ter_sum_nine_16	9	Regressor	99.90%
Gauss_blur	18	Classifier, Regressor	91.13%

For regressors, we used deep convolution neural networks [84] with a maximum of two hidden layers and trained the networks with 50 epochs. We used gradient descent to learn the hyper parameters of the network. To evaluate the goodness-of-fit for the developed models, we computed the R^2 values for each of the trained model. The R^2 is a commonly used measure to evaluate the fitness of a learning model and is defined as the following:

$$R^2 = 1 - \left(\frac{\sum_{i=1}^{N} (y_i - f_i)^2}{\sum_{i=1}^{N} (y_i - \bar{y})^2} \right) \tag{5.1}$$

where y_i represents the observed values, f_i represents the measured or predicted values, $\bar{y}$ is the mean value of the observed data, and N is the size of the dataset. In the best case, if the model can predict values exactly as the observed values, then the numerator term becomes zero resulting in an $R^2 = 1$. On the other hand a value of $R^2 = 0$ means the predictor always predicts $\bar{y}$.

The feature size, model configurations for the inference phase and the corresponding R^2 values for all the benchmarks obtained after cross validation are provided in Table 5.1. More detailed circuit characteristics of these benchmarks are also provided in Table 5.2.

5.3.2 Design space exploration via MCTS

The second phase of ML-MCTS starts with the original accelerator circuit configured as the root node. The four main steps of MCTS are then iteratively performed for a defined number of iterations. The select step finds a node in the existing tree to expand. For selection of the node, we use the classical UCT formula explained in Chapter 3 and 4. However, given the branching factor and the depth of the search tree, we observe that the selection phase should avoid expanding all the nodes before going to the next level. In AxAC synthesis, it makes more sense for the selection phase to prefer selection of nodes that lie deep in the search tree. In other words, the nodes with more approximations applied (and thus potentially more area savings). Keeping this objective in mind, we modify the selection step to consider *all* the available nodes in the search tree at once for selection instead of iterating from the root

Table 5.2: Benchmark accelerator circuits

Circuit	I/O	Candidates	Trans.†	Error metric	Area(μ^2)*	Power (mW)*
FIR filter	16/16	17	21	MRE(%)	7485.26	6.06
RGB2GRAY	8/8	5	21	MRE(%)	2427.50	0.10
Ter_sum_nine_16	16/16	8	10	MRE(%)	454.00	1.04
Gauss_blur	8/8	17	21	PSNR	7729.23	0.75

† Approximate transformations from the EvoApproxLib library [46].
* The area and power of the accelerator circuits were measured using Synopsys Design compiler using a 22nm technology library

node and stopping at the first expandable node. The latter would not let the search algorithm to explore the design space to the depth rather encourages the breadth exploration. While on the other hand, with selecting a node based on its UCT score from all available nodes in the tree would encourage selection of the nodes at the deepest level of the tree.

The selected node is then expanded if it is expandable. This is achieved by first generating a random candidate from the available candidates and then a random transformation of that type from the available approximate transformations. Without loss of generality, we use approximate transformations from the Pareto frontier set of components from EvoApproxLib [46]. The selected candidate is then replaced by the selected approximate transformation. We employ functional approximation in this work so the approximation process substitutes the C++ code of the selected candidate's operator with the C++ code of the selected approximate module. This results in a new AxAC which is added as a new node in the search tree.

The next step is error checking where the newly generated AxAC is to be checked for output error. As mentioned earlier, this is the most time dominant step since it involves circuit simulation. We avoid time-costly simulation by invoking our trained neural network model generated in the previous phase to get the error prediction. Based on the error magnitude, we see if this node represents a valid circuit configuration or an invalid one. In the latter case, we mark the node as dead node and set a negative reward for that node.

If the node is valid, we compute an area reward for the node. The reward is computed by looking at the relative area improvement as a result of the last approximation. The reward value is then updated for the all predecessor nodes. This completes one iteration of the MCTS. The DSE phase continue to grow the search tree by adding new nodes until the allocated search budget is exhausted.

5.4 Experimental results

5.4.1 Setup of experiments

To evaluate the performance and quality of results for the proposed ML-MCTS approach, we select number of benchmarks from different applications domains e.g., arithmetic, image and signal processing. These benchmarks are detailed in Table 5.2. All of these benchmarks were coded in *SystemC* since it facilitates rapid high-level design prototyping. All loops were flattened in the preprocessing step and candidates were extracted from the original accelerator circuit using the annotation provided in the source code. As the error metric, we use $PSNR$ for *Gauss_blur* and for all other benchmarks, we use *Mean Relative Error Percent* ($MRE\%$) to evaluate the quality of the approximate designs. Furthermore, we performed experiments for a gird of values for maximum error bounds of benchmarks. The error bounds for the image processing benchmark i.e., *Gauss_blur* were setup as the $PSNR$ values of $15dB$, $25dB$, $35dB$ and $45dB$ which is in line with existing works in the domain of approximate computing [24, 4, 85]. For other benchmarks, we used 0.5%, 1.0%, 2.5% and 5.0% as the maximum error bounds measured in ($MRE\%$).

The evaluation framework was coded in Python whereas for the training of the DNN models, we used TensorFlow [86], an open-source machine learning model library with Keras [87] as an interface. The experiments were performed on a compute cluster running a scientific Linux 7.2 (Nitrogen), comprising of 16 nodes with an Intel® Xeon E5-2670 @ 2.6GHz and 256 Gigabytes of main memory, of which it provides 2 Gigabytes per job. The generated approximate designs were functionally validated with datasets comprising of one million samples generated with a uniform distribution for each benchmark, except the *Gauss_blur* for which we used 500 samples from the test set of CIFAR-10 [81]. The number of candidates in Table 5.2 show only data-path elements and the circuit contributing to control-path is left exact. We used approximate transformations from an open-source approximate components library EvoApproxLib [46]. Approximate designs were synthesized using *Synopsys Design Compiler version K-2015.06* (Synopsys-DC) with a 22nm target technology library. To evaluate power consumption, we first obtained the *Switching Activity Interchange File* (SAIF) of each benchmark via simulation using their corresponding datasets and then injected the SAIFs to Synopsys-DC for measurements.

5.4.2 Results and discussion

In this subsection, the results obtained from our proposed ML-MCTS approach and the comparison with the baseline approaches are discussed. As baseline approaches, a greedy-based approach similar to [15] and an MCTS-based DSE from our previous work [57] were implemented as search methods in our framework. Both greedy and MCTS methods rely on simulations to obtain the approximation error for the intermediate designs. Moreover, these two search methods are also representatives of the

Table 5.3: Runtimes for greedy, MCTS and ML-MCTS approaches for different error bounds

BENCHMARKS	ERROR BOUNDS 0.5%	1.0%	2.5%	5.0%
GREEDY				
FIR filter	0:02:51 (25)	0:02:08 (19)	0:04:17 (36)	0:04:14 (37)
RGB2GRAY	0:02:27 (25)	0:02:25 (25)	0:02:21 (25)	0:02:28 (25)
Ter_sum_nine_16	0:02:29 (20)	0:02:19 (18)	0:02:22 (18)	0:02:34 (20)
	15 dB	***25 dB***	***35 dB***	***45 dB***
Gauss_blur	0:21:41 (76)	0:24:44 (85)	0:24:06 (85)	0:24:45 (85)
	0.5%	***1.0%***	***2.5%***	***5.0%***
MCTS [57]				
FIR filter	1:52:42 (934)	1:52:47 (944)	1:51:14 (951)	2:01:33 (987)
RGB2GRAY	1:09:03 (708)	1:08:12 (695)	1:03:24 (662)	1:00:31 (673)
Ter_sum_nine_16	2:07:02 (908)	2:01:35 (905)	1:59:09 (907)	2:01:02 (909)
	15 dB	***25 dB***	***35 dB***	***45 dB***
Gauss_blur	4:28:34 (751)	4:30:44 (840)	3:57:55 (720)	4:29:03 (847)
	0.5%	***1.0%***	***2.5%***	***5.0%***
ML-MCTS				
FIR filter	0:03:05 (936)	0:03:24 (945)	0:03:04 (938)	0:02:48 (991)
RGB2GRAY	0:01:47 (722)	0:01:43 (703)	0:01:39 (676)	0:01:39 (662)
Ter_sum_nine_16	0:02:41 (907)	0:02:32 (908)	0:02:37 (907)	0:02:31 (908)
	15 dB	***25 dB***	***35 dB***	***45 dB***
Gauss_blur	0:33:20 (838)	0:01:30 (793)	0:01:25 (821)	0:10:47 (810)

All runtimes are in h:mm:ss format

two different approaches of DSE whereas the former uses greedy heuristics to find a solution as quick as possible by selecting the best move in each iteration. The latter however requires large number of iterations to converge to a better solution since it can backtrack to different regions of the search space in different iterations and thus consumes large computational budget due to time-costly simulations. We assign the number of generations and number of random moves to the greedy method according to [15] and use a computational budget of 1000 for MCTS and ML-MCTS.

As the main objective of ML-MCTS is to speed up MCTS-based DSE, we first provide the comparison of the runtime for all three approaches. Table 5.3 provides runtime for all benchmarks against four different error bounds. The runtimes are provided in *hours:minutes:seconds* format and for each entry, the number of explored

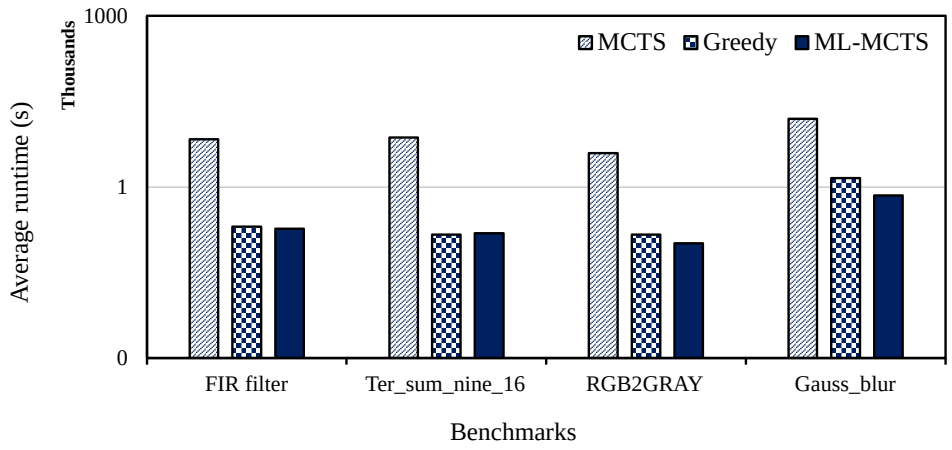


Figure 5.4: Runtime comparison for Greedy, MCTS and ML-MCTS for different benchmarks.

nodes are also shown in the brackets. For instance, the entry under 0.5% error bound for *FIR filter* for the greedy shows that it took two minutes and 51 seconds and a total of 25 nodes were explored during the DSE. It is obvious that the greedy method generally explores less number of nodes than MCTS or ML-MCTS and as a result, it could finish earlier. We set up the number of generations for greedy equal to the number of candidates, allowing the greedy method to apply approximations for all of the available candidates and therefore getting a fair chance of exploration. The number of explored nodes for MCTS and ML-MCTS are almost in the same range since they both use the same selection policy. However, the runtime of ML-MCTS is drastically shorter than MCTS since it could avoid large number of simulations by relying on the DNN-based error estimations.

The speedup can be better visualized in Figure 5.4 where the average runtime (averaged over all error bounds and for five runs) for all three approaches is depicted. Please note that the runtime is shown on a logarithmic scale. The runtime of ML-MCTS is significantly shorter than the MCTS and almost same or in some cases less than the greedy method.

Next, we compare the quality of results obtained by all three approaches. We employ a simple area saving heuristic that takes in to account the area information provided for the EvoApproxLib components library and then computes the area improvements for each approximate design generated during the DSE. Afterwards, all generated approximate designs are ranked on the basis of the area savings and the best among them is selected for the synthesis. For ML-MCTS, there is an extra step where the selected design is first validated via simulation and if it violates the error bounds, the next best design is picked and validated iteratively until finding a valid design. In Figure 5.5, the area savings results of greedy, MCTS and ML-MCTS approaches for all benchmarks against the mentioned error bounds are shown. Here, the area (normalized to the original area) of the best approximate design found by all approaches is plotted.

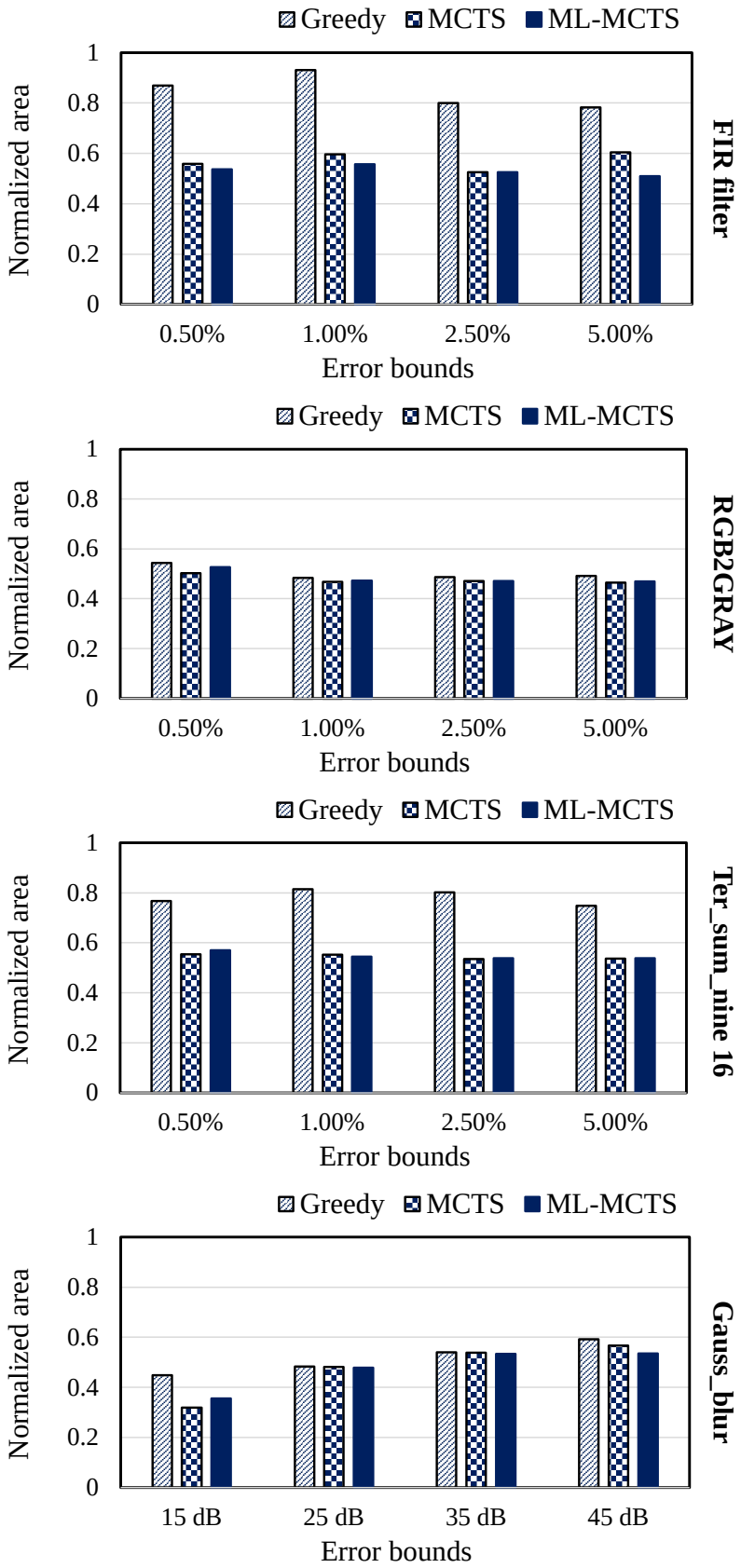


Figure 5.5: Area savings of Greedy, ML-MCTS and MCTS for different benchmarks on various error bounds.

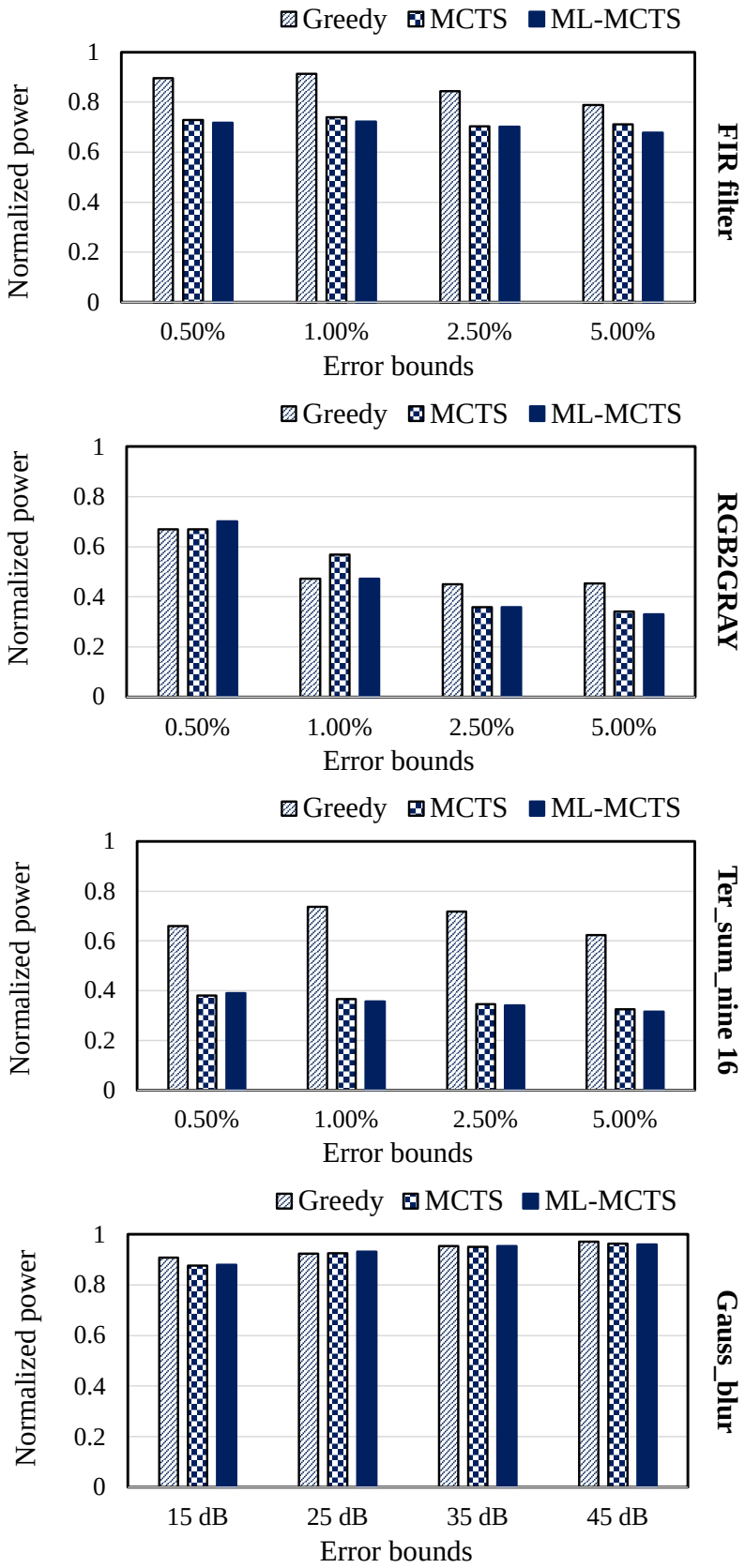


Figure 5.6: Power savings of Greedy, ML-MCTS and MCTS for different benchmarks on various error bounds.

For benchmarks *FIR filter* and *Ter_sum_nine 16*, MCTS and ML-MCTS clearly outperform greedy method in all cases by achieving significantly larger area savings. In case of *Gauss_blur* and *RGB2GRAY*, greedy could perform relatively better. Here, the area savings of greedy were slightly worse than MCTS and ML-MCTS in all cases of *RGB2GRAY* and for two cases of *Gauss_blur* i.e., 15dB and 45dB. For other two error bounds of *Gauss_blur* i.e., 25dB and 35dB, greedy, MCTS and ML-MCTS all could achieve the same area savings. At the same time, ML-MCTS was capable of achieving almost similar area savings as that of MCTS with slight variations which mostly result due to the randomness of the approximation process during the DSE phase. The maximum area saving achieved by greedy, MCTS and ML-MCTS (for *Gauss_blur*, error bound 15dB) were 55%, 68%, and 64% respectively.

Similarly we show the power savings results in Figure 5.6. Here again, the power consumption values are normalized to the original design for each benchmark. The results shown in Figure 5.6 are for the same designs shown in the area saving results. Again, for *FIR filter* and *Ter_sum_nine 16* benchmarks, MCTS and ML-MCTS achieve much better power savings than the greedy. In case of *RGB2GRAY*, greedy was able to achieve better power savings than ML-MCTS and MCTS (for error bounds of 0.5% and 1.0% respectively) whereas for other two error bounds, it could achieve less power savings than MCTS and ML-MCTS. For *Gauss_blur*, all three approaches achieve similar power savings with only one case i.e., 15dB where greedy has slightly worse power savings. The maximum power savings achieved were 55% for greedy (*RGB2GRAY* at 2.5% MRE%), 67% for MCTS (*Ter_sum_nine 16* at 5.0% MRE%) , and 68% for ML-MCTS (*Ter_sum_nine 16* at 5.0% MRE%).

5.5 Chapter conclusion

This chapter presented a fast DSE approach that leverages extremely reliable and high-speed DNN error estimation models to speed up the *Approximate Accelerator Circuit* (AxAC) synthesis framework backed by the MCTS. Using trained DNN models capable of learning the error propagation for different approximate configurations for a given benchmark, the output error can be quickly estimated without invoking the cycle-accurate simulation step. This results in huge savings of runtime and the DSE phase can explore a large number of nodes in the search space in an extremely short runtime when compared to the simulation-based DSE.

The proposed ML-MCTS approach first generates training data from the RTL simulation, deploys a DNN architecture, train it and then exploits it during the DSE phase to evaluate the error of approximate instances. It only invokes simulation in the post-processing phase to validate the AxAC instance offering best area saving. In fact, the total number of simulations performed by the ML-MCTS are far less than MCTS and even the greedy-based method. The runtime is also considerably shorter and comparable to the greedy-based method.

The proposed approach shows great potential to improve the runtime and quality of results for a search-based DSE of AxAC synthesis. However, there are a couple of avenues where it can be further enhanced. This could include for instance, studying and evaluating the ability of the proposed approach to learn the error behavior of larger designs and with larger set of approximation transformations. Moreover, the training phase can be evaluated by incorporating new architectures for the learning networks.

Chapter 6

Conclusion

This chapter concludes the thesis. It first summarizes the contributions and main outcomes of the thesis and then discusses the future work possibilities.

6.1 Summary

Approximate computing has become an effective way to obtain software / hardware implementations that offer more performance and consume less power and /or hardware area while providing acceptable results. Broadly speaking, approximate computing techniques can be applied at software or hardware level depending on the target architecture. The former achieves approximation through algorithmic techniques such as loop iteration skipping or approximate data types whereas the latter focuses on hardware logic simplification. Automated approximate accelerator synthesis techniques leverage pre-built arithmetic component substitution to systematically generate *Approximate Accelerator Circuits* (AxACs) typically via iterative search-based methods or analytical formulation of the approximation problem. Although the search-based methods are more generally applicable to a large class of circuit representation and allow more possible approximations, they are more time consuming due to exponential increase in the total number of possibilities an operator or an approximation can take. On the other hand, analytical methods could complete faster, their flexibility and scalability remains the main limitations.

In this thesis, the problem of automated AxAC synthesis has been handled with the proposed novel methodology based on an intelligent stochastic search-based optimization — *Monte Carlo Tree Search* (MCTS). MCTS has been previously applied to several other domains such as Computer Go and achieved tremendous success. However, this work attempts to use MCTS for the *Design Space Exploration* (DSE) of the approximate accelerator synthesis which is, for multiple reasons, not straight forward. The accelerator synthesis problem has longer simulation times, require efficient pruning schemes for the search space and has a different reward scheme than games. The proposed MCTS-based framework handles these challenges by adapting MCTS policies for selection, expansion and backpropagation. The proposed framework is then compared with other state-of-the-art frameworks with five practical

benchmarks where it outperformed the other techniques and achieved higher energy savings.

Later in this thesis, a hybrid methodology is proposed that handles the automated AxAC synthesis problem as a two-step process. In the first step, an analytical bit-width estimation phase generates a population of approximate designs that can represent various regions of the search space. A second step then initiates a parallel search forest rooted on the design points provided by the step one. The search forest progresses by exploring multiple regions of the search space in parallel via multiple MCTS trees. The combination of fast analytical and flexible search-based optimization results in faster exploration of the design space while achieving better quality of results. The results presented have demonstrated that the hybrid methodology is more effective than the search-based or analytical-based approaches and can achieve more area savings.

Last but not the least, Chapter 5 has demonstrated the use of high accuracy and fast deep learning-based error estimation models that are trained to learn the error propagation of the approximate instances. These models are then integrated in the MCTS-based AxAC synthesis framework to overcome the bottleneck of time-costly simulations. As a result, the DSE can perform similar number of iterations as that of simulation-based flow in a considerably short amount of time and could achieve similar area savings.

6.2 Future directions

Future directions for the improvement of the proposed work might be envisioned in multiple directions. These include enhancements for the MCTS algorithm itself e.g., adding heuristics, or looking at the DSE problem at a cross-layer level and develop a flow capable of identifying and applying approximations at multiple layers starting from the high-level. In the following, we enlist some possibilities that can be foreseen at the moment to enhance the proposed work.

1. *Heuristics for MCTS*: The problem at hand for the synthesis of AxACs tends to face explosion in terms of combinations that can be obtained during the DSE. As discussed in detail in Chapter 2 that most state-of-the-art frameworks always rely on greedy-based heuristics to prune large part of the design space. Although, it turns out to be effective in reducing the runtime, it often results in inferior performance improvements e.g., area and/or power consumption. Even with techniques like MCTS which is capable of exploring larger part of design space with a possibility of backtracking, it can be infeasible to explore the whole search space. In this regard, we introduced node pruning in Chapter 3, analytical bit-width scaling and area-based reward in Chapter 4 to help MCTS explore feasible search regions. Nevertheless, with increasing number

of candidates and available transformations during the approximation process, the design space still grows exponentially while the search algorithm is limited in terms of number of iterations. One future direction could be to investigate heuristics that can help MCTS identify regions of search space that might not lead to feasible solution so that they can be pruned early during the search process and the available search budget can be utilized to explore potentially feasible areas of the design space. Such heuristics might come from the structural information of the original circuit which can be obtained prior to the search with a preprocessing step. For instance, the components of a circuit that are closer to the output might be more sensitive to the approximations than the ones closer to the primary inputs. Alternatively, the search algorithm might spend first few iterations to sample the search space areas to gather useful error propagation information. For this, machine learning (ML) models such as ones proposed in Chapter 5 can be leveraged.

2. *Sensitivity analysis of the candidates*: For the majority of the results presented throughout this dissertation for various benchmarks, we observed that often the final AxAC identified by the search is not the one with all the candidates approximated. The approximation affects certain candidates differently as their sensitivity to the approximation varies. Of course, this has to do with the type of approximation being applied, but certain other aspects such as sensitivity of the applied input data, the statistical characteristics of the input data, the spatial relation of a candidate with other candidates inside the circuit also plays important role in this regard. We envision that a study on how these aforementioned properties affect a candidate's sensitivity to the approximation might reveal interesting results. These results can then be used to determine the optimal order for selecting candidates for approximation during the search-based optimization flow.

3. *Error masking effects*: The behavior of error propagation through the topology of any given approximate accelerator is nontrivial to model since for some arrangements of the arithmetic components, the error could get masked. Even though there has been various analytical methods such as those explained in Chapter 2 that attempt to model the error propagation, they are based on simplistic assumptions and only support combinational circuits. It might be well worth to look in detail at how the error propagation can be accurately modeled and perhaps state-of-the-art machine learning techniques could assist in capturing complex behavior of error in a circuit.

4. *Improving accuracy and confidence of predictions of ML models*: In Chapter 5, we demonstrated how the ML models trained with sufficient data can speed up the synthesis process. Although with a great potential to reduce the runtime of the overall flow, there are cases where the ML models could face decrease in prediction accuracy. This could for examples, happen in cases when there

is no sufficient training data available or if available, is highly irregular. We highlighted such a case in Chapter 5 for a *Gaussian blur filter* circuit. We also explained how we handled the inaccuracy by adding a layer of classification to refine the available training data. There are however, other possible ways to increase the accuracy or confidence of the predictions made by an ML model. Besides increasing the amount of training data, one can extract more relevant features from the hardware description of input circuit and reformat it to a representation best suited for the underlying ML models. For instance, convolution neural networks work better with high-dimensional data such as images.

5. *Enhancing the hybrid methodology*: Our hybrid methodology proposed in Chapter 4 has shown great potential for efficient DSE. In fact, with a straight forward approach based on the extreme value theory, a large part of irrelevant design space could be pruned and later the search-based optimization could spend its computational budget to explore feasible regions. We believe that the hybrid methodology could be enhanced by exploring more analytical approaches in future. To this end, we also deem ML models as an option to act as phase 1 technique in phase 1 to sample design points in the search space.

6. *Incorporating MCTS in high-level synthesis flow*: Recent years have seen an increasing interest in *High-Level Synthesis* (HLS) since it allows hardware creation at a higher level of abstraction and could be used as a mean to increase the productivity. Just recently, approximate HLS accelerators have also been attempted by some researchers. Although being able to generate hardware with less effort than the hardware description languages, still a great deal of DSE is required to produce high performance hardware with HLS. Adding approximation as one more knob requires even more extensive exploration of the design space. Furthermore, one needs to deal with the issues such as deciding loop unrolling factor, resource sharing, etc. In fact, without considering the effect of resource sharing, often the approximate accelerator might result in larger resource utilization than the original. Nonetheless, using MCTS in an HLS flow could potentially be evaluated to deal with the large design space.

7. *Cross-layer approximation flow with MCTS*: Another direction can focus on the multi-objective optimization problem at higher level of abstraction such as the algorithmic level description of the circuits. The macro blocks of the circuit can be identified, and subsequently local error budgets can be assigned to the blocks. Then, the approximate component modules can be substituted from a component library specifically characterized with the delay, power, area, and error information. Furthermore, the approach can be combined with the micro-architecture level (or even with lower levels) to obtain a cross-layer approximation flow.

List of Tables

List of Figures

Author's Publications

[1] Muhammad Awais and Marco Platzner. "MCTS-Based Synthesis Towards Efficient Approximate Accelerators". **To appear in:** *Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2021.

[2] Muhammad Awais, Hassan Ghasemzadeh Mohammadi, and Marco Platzner. "LDAX: A Learning-based Fast Design Space Exploration Framework for Approximate Circuit Synthesis". In: *Proceedings of the 31st ACM Great Lakes Symposium on VLSI (GLSVLSI)*. ACM. 2021, pp. 27–32. DOI: 10.1145/3453688.3461506.

[3] Muhammad Awais, Hassan Ghasemzadeh Mohammadi, and Marco Platzner. "A Hybrid Synthesis Methodology for Approximate Circuits". In: *Proceedings of the 30th ACM Great Lakes Symposium on VLSI (GLSVLSI)*. ACM. 2020, pp. 421–426. DOI: 10.1145/3386263.3406952.

[4] Muhammad Awais, Hassan Ghasemzadeh Mohammadi, and Marco Platzner. "An MCTS-based Framework for Synthesis of Approximate Circuits". In: *Proceedings of the 26th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE. 2018, pp. 219–224. DOI: 10.1109/VLSI-SoC.2018.8645026.

[5] Muhammad Awais. "PhD Forum: Design Space Exploration for Approximate Circuit Synthesis". *PhD Forum in 26th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. 2018. **PhD Forum Poster**.

[6] Linus Witschen, Muhammad Awais, Hassan Ghasemzadeh Mohammadi, Tobias Wiersema, and Marco Platzner. "CIRCA: Towards a Modular and Extensible Framework for Approximate Circuit Generation". In: *Microelectronics Reliability* 99 (Aug. 2019), pp. 277–290. DOI: 10.1016/j.microrel.2019.04.003.

Bibliography

[1] John M. Shalf and Robert Leland. "Computing beyond Moore's Law". In: *Computer* 48.12 (2015), pp. 14–23.

[2] Muhammad Shafique, Siddharth Garg, Jörg Henkel, and Diana Marculescu. "The EDA Challenges in the Dark Silicon Era: Temperature, Reliability, and Variability Perspectives". In: *Proceedings of the Design Automation Conference (DAC)*. ACM. 2014, pp. 1–6.

[3] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark Silicon and the End of Multicore Scaling". In: *IEEE Micro* 32.3 (2012), pp. 122–134.

[4] Sparsh Mittal. "A Survey of Techniques for Approximate Computing". In: *ACM Computing Surveys (CSUR)* 48.4 (2016), pp. 1–33.

[5] Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. "Analysis and Characterization of Inherent Application Resilience for Approximate Computing". In: *Proceedings of the Design Automation Conference (DAC)*. ACM. 2013, pp. 1–9.

[6] Swagath Venkataramani, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. "Approximate Computing and the Quest for Computing Efficiency". In: *Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE. 2015, pp. 1–6.

[7] Honglan Jiang, Cong Liu, Leibo Liu, Fabrizio Lombardi, and Jie Han. "A Review, Classification, and Comparative Evaluation of Approximate Arithmetic Circuits". In: *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13.4 (2017), pp. 1–34.

[8] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. "Managing Performance vs. Accuracy Trade-offs with Loop Perforation". In: *Proceedings of the ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM. 2011, pp. 124–134.

[9] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D Nguyen. "Approxhadoop: Bringing Approximations to Mapreduce Frameworks". In: *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM. 2015, pp. 383–397.

[10] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. "AxNN: Energy-efficient Neuromorphic Systems using Approximate Computing". In: *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE. 2014, pp. 27–32.

[11] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. "EnerJ: Approximate Data Types for Safe and General Low-power Computation". In: *ACM SIGPLAN Notices* 46.6 (2011), pp. 164–174.

[12] Amir Yazdanbakhsh, Divya Mahajan, Bradley Thwaites, Jongse Park, Anandhavel Nagendrakumar, Sindhuja Sethuraman, Kartik Ramkrishnan, Nishanthi Ravindran, Rudra Jariwala, Abbas Rahimi, et al. "Axilog: Language Support for Approximate Hardware Design". In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2015, pp. 812–817.

[13] Jorge Castro-Godinez, Sven Esser, Muhammad Shafique, Santiago Pagani, and Jörg Henkel. "Compiler-driven Error Analysis for Designing Approximate Accelerators". In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018, pp. 1027–1032.

[14] Arun Chandrasekharan, Mathias Soeken, Daniel Groesse, and Rolf Drechsler. "Approximation-aware Rewriting of AIGs for Error Tolerant Applications". In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. ACM, 2016, pp. 1–8.

[15] Kumud Nepal, Yueting Li, R Iris Bahar, and Sherief Reda. "ABACUS: A Technique for Automated Behavioral Synthesis of Approximate Computing Circuits". In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2014, pp. 1–6.

[16] Kumud Nepal, Soheil Hashemi, Hokchhay Tann, R Iris Bahar, and Sherief Reda. "Automated High-level Generation of Low-power Approximate Computing Circuits". In: *IEEE Transactions on Emerging Topics in Computing* 7.1 (2016), pp. 18–30.

[17] Ashish Ranjan, Arnab Raha, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. "ASLAN: Synthesis of Approximate Sequential Circuits". In: *Proceedings of the Design, Automation & Test Conference in Europe (DATE)*. IEEE, 2014, pp. 1–6.

[18] Linus Witschen, Hassan Ghasemzadeh Mohammadi, Matthias Artmann, and Marco Platzner. "Jump search: A Fast Technique for the Synthesis of Approximate Circuits". In: *Proceedings of the ACM Great Lakes Symposium on VLSI (GLSVLSI)*. ACM. 2019, pp. 153–158.

[19] Ilaria Scarabottolo, Giovanni Ansaloni, and Laura Pozzi. "Circuit Carving: A Methodology for the Design of Approximate Hardware". In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018, pp. 545–550.

[20] Jeremy Schlachter, Vincent Camus, Krishna V. Palem, and Christian Enz. "Design and Applications of Approximate Circuits by Gate-Level Pruning". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.5 (2017), pp. 1694–1702.

[21] Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. "Substitute-and-simplify: A Unified Design Paradigm for Approximate and Quality Configurable Circuits". In: *Proceedings of the Design, Automation & Test Conference in Europe (DATE)*. IEEE, 2013, pp. 1367–1372.

[22] Soheil Hashemi, Hokchhay Tann, and Sherief Reda. "BLASYS: Approximate Logic Synthesis using Boolean Matrix Factorization". In: *Proceedings of the Design Automation Conference (DAC)*. ACM. 2018, p. 55.

[23] Gai Liu and Zhiru Zhang. "Statistically Certified Approximate Logic Synthesis". In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 344–351.

[24] Vaibhav Gupta, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy. "IMPACT: Imprecise Adders for Low-power Approximate Computing". In: *Proceedings of the IEEE/ACM International Symposium on Low-power Electronics and Design (ISLPED)*. IEEE, 2011, pp. 409–414.

[25] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas. "Bio-inspired Imprecise Computational Blocks for Efficient VLSI Implementation of Soft Computing Applications". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 57.4 (2010), pp. 850–862.

[26] Parag Kulkarni, Puneet Gupta, and Miloš D Ercegovac. "Trading Accuracy for Power in a Multiplier Architecture". In: *Journal of Low Power Electronics* 7.4 (2011), pp. 490–501.

[27] Swagath Venkataramani, Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. "Quality Programmable Vector Processors for Approximate Computing". In: *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2013, pp. 1–12.

[28] Omid Akbari, Mehdi Kamal, Ali Afzali-Kusha, Massoud Pedram, and Muhammad Shafique. "X-CGRA: An Energy-Efficient Approximate Coarse-Grained Reconfigurable Architecture". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.10 (2019), pp. 2558–2571.

[29] Mohsen Imani, Abbas Rahimi, and Tajana S Rosing. "Resistive Configurable Associative Memory for Approximate Computing". In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2016, pp. 1327–1332.

[30] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. "Load Value Approximation". In: *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2014, pp. 127–139.

[31] Ashish Ranjan, Swagath Venkataramani, Zoha Pajouhi, Rangharajan Venkatesan, Kaushik Roy, and Anand Raghunathan. "STAxCache: An Approximate, Energy Efficient STT-MRAM Cache". In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2017, pp. 356–361.

[32] Arnab Raha, Soubhagya Sutar, Hrishikesh Jayakumar, and Vijay Raghunathan. "Quality Configurable Approximate DRAM". In: *IEEE Transactions on Computers* 66.7 (2016), pp. 1172–1187.

[33] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. "Architecture Support for Disciplined Approximate Programming". In: *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS))*. ACM. 2012, pp. 301–312.

[34] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. "Neural Acceleration for General-purpose Approximate Programs". In: *Communications of the ACM* 58.1 (2014), pp. 105–115.

[35] Ajay K Verma, Philip Brisk, and Paolo Ienne. "Variable Latency Speculative Addition: A New Paradigm for Arithmetic Circuit Design". In: *Proceedings of the Design, Automation and Test in Europe (DATE)*. IEEE. 2008, pp. 1250–1255.

[36] Ning Zhu, Wang Ling Goh, Weija Zhang, Kiat Seng Yeo, and Zhi Hui Kong. "Design of Low-power High-speed Truncation-error-tolerant Adder and its Application in Digital Signal Processing". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 18.8 (2010), pp. 1225–1229.

[37] Andrew B Kahng and Seokhyeong Kang. "Accuracy-configurable Adder for Approximate Arithmetic Designs". In: *Proceedings of the Design Automation Conference (DAC)*. ACM. 2012, pp. 820–825.

[38] Muhammad Shafique, Waqas Ahmad, Rehan Hafiz, and Jörg Henkel. "A Low Latency Generic Accuracy Configurable Adder". In: *Proceedings of the Design Automation Conference (DAC)*. IEEE. 2015, pp. 1–6.

[39] Kartikeya Bhardwaj, Pravin S Mane, and Jörg Henkel. "Power and Area Efficient Approximate Wallace Tree Multiplier for Error-resilient Systems". In: *Proceedings of the International Symposium on Quality Electronic Design (ISQED)*. IEEE. 2014, pp. 263–269.

[40] Cong Liu, Jie Han, and Fabrizio Lombardi. "A Low-power, High-performance Approximate Multiplier with Configurable Partial Error Recovery". In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2014, pp. 1–4.

[41] Amir Momeni, Jie Han, Paolo Montuschi, and Fabrizio Lombardi. "Design and Analysis of Approximate Compressors for Multiplication". In: *IEEE Transactions on Computers* 64.4 (2015), pp. 984–994.

[42] Bharath Srinivas Prabakaran, Semeen Rehman, Muhammad Abdullah Hanif, Salim Ullah, Ghazal Mazaheri, Akash Kumar, and Muhammad Shafique. "DeMAS: An Efficient Design Methodology for Building Approximate Adders for Fpga-based Systems". In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018, pp. 917–920.

[43] Semeen Rehman, Walaa El-Harouni, Muhammad Shafique, Akash Kumar, Jorg Henkel, and Jörg Henkel. "Architectural-space Exploration of Approximate Multipliers". In: *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2016, pp. 1–8.

[44] Salim Ullah, Sanjeev Sripadraj Murthy, and Akash Kumar. "SMApproxLib: Library of FPGA-based Approximate Multipliers". In: *Proceedings of the ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE. 2018, pp. 1–6.

[45] Muhammad Abdullah Hanif, Rehan Hafiz, Osman Hasan, and Muhammad Shafique. "QuAd: Design and Analysis of Quality-Area Optimal Low-Latency Approximate Adders". In: *Proceedings of the Design Automation Conference (DAC)*. IEEE. 2017, pp. 1–6.

[46] Vojtech Mrazek, Radek Hrbacek, Zdenek Vasicek, and Lukas Sekanina. "EvoApproxSb: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods". In: *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*. IEEE, 2017, pp. 258–261.

[47] Jiawei Huang, John Lach, and Gabriel Robins. "A Methodology for Energy-quality Tradeoff using Imprecise Hardware". In: *Proceedings of the Design Automation Conference (DAC)*. ACM. 2012, pp. 504–509.

[48] Cong Liu, Jie Han, and Fabrizio Lombardi. "An Analytical Framework for Evaluating the Error Characteristics of Approximate Adders". In: *IEEE Transactions on Computers* 64.5 (2015), pp. 1268–1281.

[49] Deepashree Sengupta, Farhana Sharmin Snigdha, Jiang Hu, and Sachin S Sapatnekar. "SABER: Selection of Approximate Bits for the Design of Error Tolerant Circuits". In: *Proceedings of the Design Automation Conference (DAC)*. ACM. 2017, pp. 1–6.

[50] Deepashree Sengupta, Farhana Sharmin Snigdha, Jiang Hu, and Sachin S. Sapatnekar. "An Analytical Approach for Error PMF Characterization in Approximate Circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.1 (2019), pp. 70–83.

[51] Mario Barbareschi, Federico Iannucci, and Antonino Mazzeo. "Automatic Design Space Exploration of Approximate Algorithms for Big Data Applications". In: *Proceedings of the International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. IEEE. 2016, pp. 40–45.

[52] Swagath Venkataramani, Amit Sabne, Vivek Kozhikkottu, Kaushik Roy, and Anand Raghunathan. "SALSA: Systematic Logic Synthesis of Approximate Circuits". In: *Proceedings of the Design Automation Conference (DAC)*. ACM, 2012, pp. 796–801.

[53] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. "Accept: A Programmer-guided Compiler Framework for Practical Approximate Computing". In: *University of Washington Technical Report UW-CSE-15-01* 1.2 (2015).

[54] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. "ASAC: Automatic Sensitivity Analysis for Approximate Computing". In: *ACM SIGPLAN Notices* 49.5 (2014), pp. 95–104.

[55] Seogoo Lee, Dongwook Lee, Kyungtae Han, Emily Shriver, Lizy K John, and Andreas Gerstlauer. "Statistical Quality Modeling of Approximate Hardware". In: *Proceedings of the International Symposium on Quality Electronic Design (ISQED)*. IEEE. 2016, pp. 163–168.

[56] Linus Witschen, Muhammad Awais, Hassan Ghasemzadeh Mohammadi, Tobias Wiersema, and Marco Platzner. "CIRCA: Towards a Modular and Extensible Framework for Approximate Circuit Generation". In: *Microelectronics Reliability* 99 (Aug. 2019), pp. 277–290. DOI: 10.1016/j.microrel.2019.04.003.

[57] Muhammad Awais, Hassan Ghasemzadeh Mohammadi, and Marco Platzner. "An MCTS-based Framework for Synthesis of Approximate Circuits". In: *Proceedings of the 26th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE. 2018, pp. 219–224. DOI: 10.1109/VLSI-SoC.2018.8645026.

[58] Zdenek Vasicek. "Relaxed Equivalence Checking: A New Challenge in Logic Synthesis". In: *Proceedings of the International Symposium on Design and Diagnostics of Electronic Circuits & Systems*. IEEE, 2017, pp. 1–6.

[59] Ilaria Scarabottolo, Giovanni Ansaloni, George A Constantinides, Laura Pozzi, and Sherief Reda. "Approximate Logic Synthesis: A Survey". In: *Proceedings of the IEEE* 108.12 (2020), pp. 2195–2213.

[60] Arun Chandrasekharan, Mathias Soeken, Daniel Große, and Rolf Drechsler. "Precise Error Determination of Approximated Components in Sequential Circuits with Model Checking". In: *Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE. 2016, pp. 1–6.

[61] Siyuan Xu and Benjamin Carrion Schafer. "Exposing Approximate Computing Optimizations at Different Levels: From Behavioral to Gate-Level". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.11 (2017), pp. 3077–3088.

[62] Vojtech Mrazek, Muhammad Abdullah Hanif, Zdenek Vasicek, Lukas Sekanina, and Muhammad Shafique. "AutoAx: An Automatic Design Space Exploration and Circuit Building Methodology utilizing Libraries of Approximate Components". In: *Proceedings of the Design Automation Conference (DAC)*. IEEE. 2019, pp. 1–6.

[63] Shinya Takamaeda-Yamazaki. "Pyverilog: A Python-based Hardware Design Processing Toolkit for Verilog HDL". In: *Proceedings of the International Symposium on Applied Reconfigurable Computing*. Springer. 2015, pp. 451–460.

[64] Ben James Winer. *Statistical Principles in Experimental Design.* McGraw-Hill Book Company, 1962.

[65] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. "A survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43.

[66] Bilal Kartal, John Koenig, and Stephen J Guy. "User-driven Narrative Variation in Large Story Domains using Monte Carlo Tree Search". In: *Proceedings of the International Conference on Autonomous Agents and Multi-agent Systems*. International Foundation for Autonomous Agents and Multiagent Systems. 2014, pp. 69–76.

[67] Levente Kocsis and Csaba Szepesvári. "Bandit based Monte-Carlo Planning". In: *Proceedings of the European Conference on Machine Learning*. Springer. 2006, pp. 282–293.

[68] Robert Brayton and Alan Mishchenko. "ABC: An academic industrial-strength verification tool". In: *Proceedings of the International Conference on Computer Aided Verification*. Springer. 2010, pp. 24–40.

[69] Clifford Wolf. *Yosys Open SYnthesis Suite*. `http://www.clifford.at/yosys/`. Accessed: 2017-05-16.

[70] James E Doran and Donald Michie. "Experiments with the Graph Traverser Program". In: *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* 294.1437 (1966), pp. 235–259.

[71] Edmund K Burke and Yuri Bykov. "The Late Acceptance Hill-climbing Heuristic". In: *European Journal of Operational Research* 258.1 (2017), pp. 70–78.

[72] Peter JM Van Laarhoven and Emile HL Aarts. *Simulated Annealing*. Springer, 1987, pp. 7–15.

[73] Guido van Rossum, Barry Warsaw, and Nick Coghlan. *PEP 8 - Style Guide for Python Code*. `https://www.python.org/dev/peps/pep-0008/`. Accessed: 2018-08-28.

[74] Ben Reynwar. *Decimation-In-Time fast Fourier Transform*. `https://github.com/benreynwar/fft-dit-fpga`. Accessed: 2018-09-05.

[75] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, et al. "VTR 7.0: Next Generation Architecture and CAD System for FPGAs". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 7.2 (2014), pp. 1–30.

[76] Altera Corporation. *Altera Advanced Synthesis Cookbook*. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/stx_cookbook.pdf`. Accessed: 2018-09-05.

[77] David Lundgren. *OpenCores jpegencode*. `https://github.com/chiggs/oc_jpegencode`. Accessed: 2017-07-25.

[78] Alexander J McNeil. "Estimating the Tails of Loss Severity Distributions using Extreme Value Theory". In: *ASTIN Bulletin: The Journal of the IAA* 27.1 (1997), pp. 117–137.

[79] Enrique Castillo and Ali S Hadi. "Fitting the Generalized Pareto Distribution to Data". In: *Journal of the American Statistical Association* 92.440 (1997), pp. 1609–1620.

[80] Emre Özer, Andy P Nisbet, and David Gregg. "Stochastic Bit-width Approximation using Extreme Value Theory for Customizable Processors". In: *Proceedings of the International Conference on Compiler Construction*. Springer. 2004, pp. 250–264.

[81] *The CIFAR-10 dataset*. `https://www.cs.toronto.edu/~kriz/cifar.html`. Accessed: 2021-02-05.

[82] Leo Breiman. "Random Forests". In: *Machine Learning* 45.1 (2001), pp. 5–32.

[83] Tong Zhang. "Solving Large Scale Linear Prediction Problems using Stochastic Gradient Descent Algorithms". In: *Proceedings of the International Conference on Machine Learning*. ACM. 2004, pp. 116–123.

[84] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521.7553 (2015), pp. 436–444.

[85] Siyuan Xu and Benjamin Carrion Schafer. "On The Design of High Performance HW Accelerator through High-level Synthesis Scheduling Approximations". In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2020, pp. 1378–1383.

[86] *TensorFlow*. `https://www.tensorflow.org`. Accessed: 2021-02-23.

[87] *Keras API*. `https://keras.io/`. Accessed: 2021-02-23.