AppAxO: Designing Application-specific Approximate Operators for FPGA-based Embedded Systems

SALIM ULLAH, SIVA SATYENDRA SAHOO, NEMATH AHMED, DEBABRATA CHAUDHURY, and AKASH KUMAR, The Chair for Processor Design, Center for Advancing Electronics Dresden (CfAED), Technische Universität Dresden, Germany

Approximate arithmetic operators, such as adders and multipliers, are increasingly used to satisfy the energy and performance requirements of resource-constrained embedded systems. However, most of the available approximate operators have an application-agnostic design methodology, and the efficacy of these operators can only be evaluated by employing them in the applications. Further, the various available libraries of approximate operators do not share any standard approximation-induction policy to design new operators according to an application's accuracy and performance constraints. These limitations also hinder the utilization of machine learning models to explore and determine approximate operators according to an application's requirements. In this work, we present a generic design methodology for implementing FPGA-based application-specific approximate arithmetic operators. Our proposed technique utilizes lookup tables and carry-chains of FPGAs to implement approximate operators according to the input configurations. For instance, for an M × N accurate multiplier utilizing K lookup tables, our methodology utilizes K-bit configurations to design 2^K approximate multipliers. We then utilize various machine learning models to evaluate and select configurations satisfying application accuracy and performance constraints. We have evaluated our proposed methodology for three benchmark applications, i.e., biomedical signal processing, image processing, and ANNs. We report more non-dominated approximate multipliers with better hypervolume contribution than state-of-the-art designs for these benchmark applications with the proposed design methodology.

CCS Concepts: • **Computer systems organization → Embedded hardware**; • **Hardware → Hardware accelerators**; **Arithmetic and datapath circuits**.

Additional Key Words and Phrases: Approximate Computing, Arithmetic Circuits, Multipliers, Machine Learning Models, Energy Efficient Computing, FPGA, High-level Synthesis

ACM Reference Format:
Salim Ullah, Siva Satyendra Sahoo, Nemath Ahmed, Debabrata Chaudhury, and Akash Kumar. 2022. *AppAxO*: Designing Application-specific Approximate Operators for FPGA-based Embedded Systems . *ACM Trans. Embedd. Comput. Syst.* 1, 1 (February 2022), 31 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A large number of applications in the domain of signal processing, computer vision, and machine learning possess an inherent error resilience [7]. These applications can produce acceptable quality results despite some inexactness in the representation of their processed data and corresponding operations. Further, the error-tolerant computations in these

Authors' address: Salim Ullah, salim.ullah@tu-dresden.de; Siva Satyendra Sahoo, siva_satyendra.sahoo@tu-dresden.de; Nemath Ahmed, nemath.ahmed@mailbox.tu-dresden.de; Debabrata Chaudhury, debabrata.chaudhury@mailbox.tu-dresden.de; Akash Kumar, The Chair for Processor Design, Center for Advancing Electronics Dresden (CfAED), Technische Universität Dresden , Dresden, Germany, akash.kumar@tu-dresden.de.

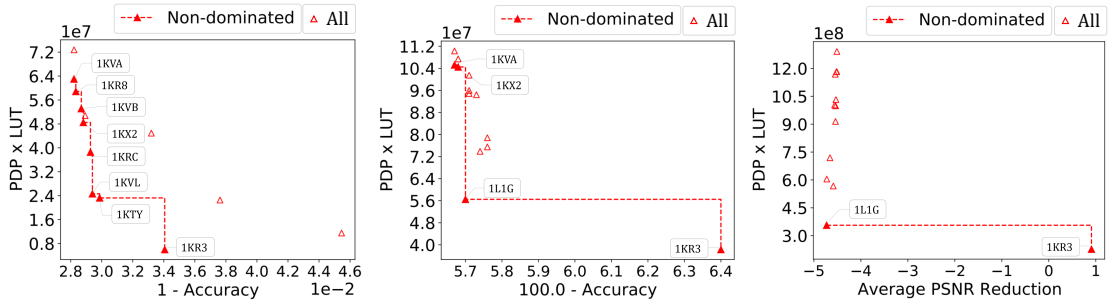


Fig. 1. Application-level performance-accuracy analysis of utilizing approximate multipliers from [21] for three different applications (a) ECG QRS peak detection (b) MLP for MNIST dataset classification (c) Gaussian image smoothing filter

applications are often the major contributors to their respective implementations' overall resource utilization, latency, and energy consumption [38]. For such applications, the approximate computing paradigm has emerged as a viable solution for implementing high-performance and energy-efficient hardware accelerators. Approximate computing methods bargain the computational accuracy of an application to achieve performance gains. These methods can be applied at any layer of the computation stack [19].

Among these layers of approximation, circuit-level techniques have been a major focus of research for resource-constrained embedded systems [10, 12, 15, 21–23, 28–30, 32, 34, 36]. However, the state-of-the-art approximate arithmetic blocks have two main limitations:

1) These approximate blocks lack a consistent design methodology and have considered different strategies for introducing approximations to obtain performance gains. For example, the authors of [36] and [32] have considered the techniques of *least-significant product bit truncation* and *elimination of carry propagation among product bits*, respectively, for designing 4×4 approximate multipliers for Field Programmable Gate Array (FPGA)-based systems. Further, for a fixed n-bit operator, most of these blocks provide a fixed output accuracy and corresponding performance gain. For example, compared to a 4×4 accurate multiplier, the works in [36] and [32] report a 63% and 25% reduction in the resource utilization, respectively, and 0.55 and 0.072 degradation in the output (*average relative error*), respectively. Therefore, to design a new approximate block with a modified accuracy-performance constraint, these techniques cannot be utilized and calls for the exploration of some other approximation techniques. The modular design methodology, as used in [29] and [15], of designing N × N approximate multipliers from $\frac{N}{2}$ approximate multipliers provide a limited design space and may not be sufficient to satisfy the modified accuracy-performance constraint.

2) Most of the state-of-the-art approximate arithmetic operators are designed without considering an application's accuracy-performance constraints. The application agnostic-design methodology can result in approximate operators which may not satisfy an application's accuracy-performance constraints. For example, Figure 1 presents the performance-accuracy analysis for three different applications[1] using the approximate signed multipliers library *EvoApprox*[2] from [21, 23]. We have used the Power-Delay Product (PDP) and resource utilization for reporting the corresponding performance of each application by implementing it on Xilinx UltraScale FPGA using Xilinx Vivado. For PDP, the dynamic power is computed in μW, and the critical path delay (CPD) is reported in ns. Further, we have considered

[1]The reported accuracy metric in ECG application varies between 0 and 1 unlike for MNIST classification which varies between 0 and 100%.
[2]https://ehw.fit.vutbr.cz/evoapproxlib/?folder=multiplers/8x8_signed

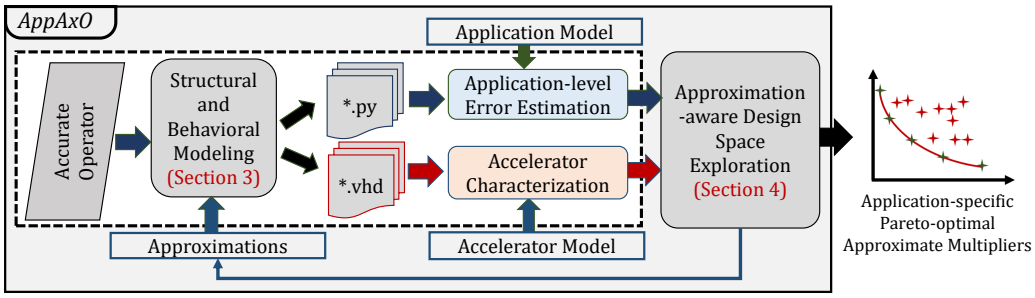


Fig. 2. Proposed framework for *AppAxO*

the 6-input Lookup Tables to report resource utilization. Similarly, we have used Python-based models to report each application's output accuracy for various approximate multipliers. These results identify the approximate multipliers which contribute to non-dominated hardware accelerators for each application. Figure 1(a) describes the hardware accelerator results for the QRS peak detection in Electrocardiographic (ECG) signals. To report the application's output accuracy for various approximate multipliers, we have used the *accuracy of the peak detection* metric. The results show that 9 different approximate multipliers (out of 13) contribute to non-dominated design points. Figure 1(b) shows the implementation results of deploying the 13 approximate multipliers in the last layer of a lightweight Multilayer Perceptron (MLP) to classify the Modified National Institute of Standards and Technology (MNIST) dataset [17]. The application-agnostic design of the multipliers has resulted in only 4 non-dominated design points (accelerator designs). As shown in Figure 1(c), this behavior is exacerbated in the implementation of the Gaussian Image Smoothing filter using the approximate multipliers. The performance and the accuracy results (for 45 images) show that only two approximate multipliers contribute to the non-dominated design points. Moreover, the application-level analysis also reveals that the accurate multiplier *1KV8* from the 'EvoApprox' library [21, 23] does not contribute to any non-dominated accelerator design point for any application.

Therefore it is necessary to define a generic and optimized design methodology that can generate application-specific approximate operators satisfying its accuracy-performance constraints. Towards this end, we present *AppAxO*: a methodology for designing application-specific approximate arithmetic operators for FPGAs-based systems in this work. As shown in Figure 2, the proposed methodology involves circuit-level modeling and novel Design Space Exploration (DSE) methods for fast design of approximate arithmetic operators that can leverage the inherent robustness of error-tolerant applications. We have considered multipliers and adders as example operators to discuss our methodology and present our results; however, the proposed methodology is generic and can be used for designing any *soft logic-based* operators including dividers. Our proposed implementations utilize the 6-input Lookup table (LUT) and associated carry chains of modern FPGAs building blocks. Our novel contributions include:

- *A systematic methodology for approximate operators generation:* We provide a systematic and generic methodology for implementing approximate operators of arbitrary size for FPGA-based systems. Our methodology utilizes the 6-input LUTs and the associated carry chains of FPGAs to implement approximate operators according to input configuration. For instance, the input configurations—a binary string—identify the LUTs, in an accurate operator implementation, that should be disabled to realize a corresponding approximate operator. For an M × N accurate multiplier, utilizing 'K' LUTs, our methodology provides 2^K approximate multipliers with different accuracy and performance parameters.

- *Application-specific multiplier configurations:* We utilize a Multi-objective Bayesian Optimization (MBO)-based exploration method to generate only those approximate multiplier configurations (a binary string) that satisfy an application's accuracy and performance constraints. Our proposed multiplier generation methodology uses these configurations to implement the respective multipliers for the application.
- *An efficient DSE methodology:* We utilize various Machine Learning (ML) models to propose a genetic algorithm-based DSE methodology. Our methodology deploys various ML models to explore the large design space of individual multipliers and their utilization in various applications by estimating the behavioral accuracy and corresponding performance gains.

The rest of the article is organized as follows. We provide a brief overview of the relevant background and related works in Section 2. Section 3 describes the systematic modeling methodology used for designing arbitrary approximate multipliers. The DSE methods adopted for designing application-specific approximate multipliers is described in Section 4. In Section 5, the results from the experimental evaluation of the proposed framework are presented, followed by a discussion on the scope of related future research in Section 6.

2 BACKGROUND AND RELATED WORKS

2.1 Approximate Computing

Multiplication, being one of the computationally complex and intensively used instructions in various application domains, has remained the focus of many recent approximate computing works [10, 14–16, 21–23, 26, 27, 29, 32–36]. These works have utilized various techniques, such as truncation and the design of inexact hardware, to realize approximate multipliers. However, most of these works have focused on proposing single designs of approximate multipliers. For example, some works utilize various truncation techniques to produce an N-bit output for an N × N multiplier [14, 26]. Some works perform quantization (truncation) of the operands to employ an M × M multiplier for implementing an N × N multiplier where $M < N$ [13, 16]. Similarly, some works focus on approximating various sub-operations of the final product computation. For example, the works in [15] and [29] have presented 2×2 approximate unsigned multiplier blocks for ASICs. These designs are based on the simplified, and approximate Karnaugh maps to reduce multiplication logic. The authors of [34] present 4×2 and 4×4 approximate unsigned multiplier designs for FPGA-based systems. These designs utilize the truncation of least significant product bits and approximate addition of generated partial products to reduce the total number of utilized LUTs. The authors of [36] have proposed approximate radix-4 Booth-based approximate signed multipliers for FPGA-based systems. Their proposed technique deploys the truncation of the least significant partial product bit in each generated partial product to compute the final product. Further, they have also shown that utilizing an approximate M × M multiplier to implement a precision-reduced approximate N × N multiplier, where $M < N$, can significantly improve the overall performance of the multiplier. The work in [21] utilizes various previously proposed approximate multipliers and adders to present a library of 8×8 approximate unsigned multipliers referred to as 'EvoApprox'. In their follow-up work, the authors of [21] have extended the EvoApprox library to include signed multipliers [23]. Considering 4×4 as an elementary module, the work in [32] has proposed 3 different 4×4 approximate multiplier designs to propose a library of approximate multipliers for FPGA-based systems. These approximate designs are based on the approximate and parallel generation of all product bits. For this purpose, the carry propagation between the product bits is eliminated.

As discussed in Section 1, these designs do not share a consistent design technique and follow an application-agnostic methodology for designing approximate multipliers. The work presented in [22] has used a Cartesian Genetic

Table 1. Comparing related works

Article	Platform	Focus	Application Specific Operators	Constraints for New Operators/DSE			
				Accuracy	Resources	Latency	Power
P. Kulkarni [15]	ASIC	Single Multiplier	✗	✗	✗	✗	✗
S. Rehman [29]	ASIC	Single Multiplier + DSE	✗	✓	✓	✗	✓
S. Ullah [34]	FPGA	Two Multipliers	✗	✗	✗	✗	✗
S. Ullah [36]	FPGA	Single Multiplier + DSE	✗	✓	✓	✗	✗
V. Mrazek [21]	ASIC	Library of Multipliers and adders	✗	✗	✗	✗	✗
S. Ullah [32]	FPGA	Library of Multipliers	✗	✗	✗	✗	✗
V. Mrazek [22]	ASIC	Library of Multipliers + DSE	✓	✓	✓	✗	✗
V. Mrazek [20]	ASIC	DSE	✗	✓	✓	✓	✓
AppAxO	FPGA	Library of Multipliers + DSE	✓	✓	✓	✓	✓

Programming (CGP)-based technique to design unsigned approximate multipliers for Artificial Neural Network (ANN). This technique utilizes a $2D$ array of 2-input logic gates to represent multipliers. The initial population for their technique consists of 3 accurate and a few approximate unsigned multipliers. In each iteration of the CGP, a new set of multipliers is generated according to a predefined approximation error ϵ. The multipliers' efficacy is assessed by deploying these multipliers in an ANN and evaluating the network's output accuracy after retraining. Depending upon the network's output accuracy, the approximation error value ϵ can be adjusted for the next iteration of the CGP. However, this technique does not consider multiplier performance parameters, such as critical path delay and dynamic power, while generating new approximate multipliers. The output accuracy of the network is the only factor deciding the generation of an approximate multiplier. Further, the generated approximate multipliers are unsigned, and separate circuitry for calculating the product sign has been used to deploy ANNs.

2.2 DSE for Approximate Computing

The work presented in [22] generates new approximate unsigned multipliers during the DSE for ANNs. However, some recent works have also utilized various machine learning techniques for performing application-specific DSE on existing approximate arithmetic operators. For example, the authors of [20] have used various machine learning models to perform an efficient DSE for Sobel filter considering approximate adders. In their proposed technique, the non-feasible approximate adders (from an existing library of approximate adders) are filtered out by computing the weighted mean error distance and hardware performance metrics of each approximate circuit. The feasible approximate adders are then used to train machine learning models to estimate the behavioral accuracy and performance parameters of the Sobel filter accelerator. The authors of [36] have also used Genetic Algorithms (GA)-based multi-objective design space exploration for Gaussian smoothing filter. Utilizing a set of an accurate and an approximate multipliers, they have used GA to find a feasible combination of multiplier types with a trade-off between output quality and LUT utilization. The authors of [29] have also utilized a depth-first search-based methodology to perform architectural space exploration for designing larger multipliers from 2×2 approximate multipliers. Their technique utilizes a weighted average of the area and power to identify feasible design points.

Applications from different domains exhibit diverse error-tolerance for approximate arithmetic operators and hold different output accuracy and performance (resources, latency, and power) requirements. However, the various related state-of-the-art works, summarized in Table 1, do not offer a methodology for generating approximate operators according to an application's accuracy and performance constraints. To address the limitations of the state-of-the-art works, we present *AppAxO* methodology to generate new approximate arithmetic operators according to an application's

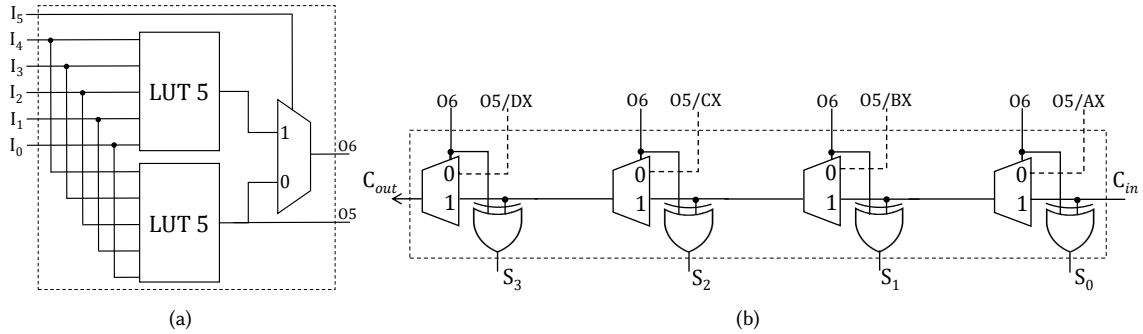


Fig. 3. Combinational logic block of modern FPGAs: (a) 6-input LUT (b) 4-bit wide carry chain

accuracy and performance constraints. Towards this end, *AppAxO* utilizes various machine learning models to identify feasible approximate operators for an input application efficiently. Our proposed methodology is generic and can generate approximate circuits for any arithmetic operator which utilizes LUTs and carry chains for its implementation. It should be noted that *AppAxO* focuses on circuit-level approximations only. Other optimization techniques, such as exploring the impact of HLS directives and approximations across multiple layers of computation stack, are orthogonal to our proposed work. These optimization techniques can be integrated with our proposed methodology.

3 MODELING APPROXIMATE ARITHMETIC OPERATORS

AppAxO proposes a systematic methodology for designing approximate arithmetic operators from the corresponding accurate implementations of the operators. The accurate operators are implemented by utilizing the 6-input LUTs, and the associate carry chains of FPGAs. For example, Figure 3 describes the structure of a 6-input LUT and a 4-bit wide carry chain in a Configurable Logic Blocks (CLB) of an FPGA [37]. The 6-input LUT, shown in Figure 3(a), utilizes two 5-input LUTs to implement either two distinct 5-input combinational circuits or a single 6-input combinational circuit. The functionality of a LUT is defined by providing it a 64-bit INIT value. The outputs of LUTs, $O5$ and $O6$, are also used to control the associated carry chain in the CLB, as shown in Figure 3(b). The O5 and O6 signals are used as the carry-generate and carry-propagate signals, respectively. The CLB also allows bypassing the $O5$ signal and providing an external signal to the carry chain. The carry-out C_{out} of a carry chain in a CLB can be interfaced with the C_{in} of a carry chain in another CLB. Each pair of a *MUX* and an *XOR gate* in the carry chain represents a carry chain cell. For example, Figure 3(b) shows a 4-bit wide carry chain with four carry chain cells. In this work, we use accurate multipliers and adders to present the *AppAxO* methodology for implementing approximate multipliers and adders. Specifically, we enable better utilization of the FPGA resources (LUTs and carry chains), resulting in higher packing efficiency (CLB utilization) than state-of-the-art works [27].

$$P = a_{N-1}b_{M-1}2^{N+M-2} + \sum_{n=0}^{N-2}\sum_{m=0}^{M-2} a_n b_m 2^{n+m} + 2^{N-1}\sum_{m=0}^{M-2}\overline{a_{N-1}b_m}2^m + 2^{M-1}\sum_{n=0}^{N-2}\overline{b_{M-1}a_n}2^n + 2^{N-1} + 2^{M-1} + 2^{N+M-1} \quad (1)$$

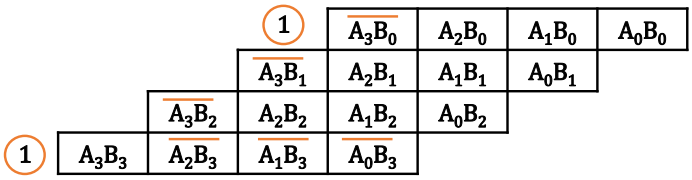


Fig. 4. 4×4 accurate signed multiplier partial products

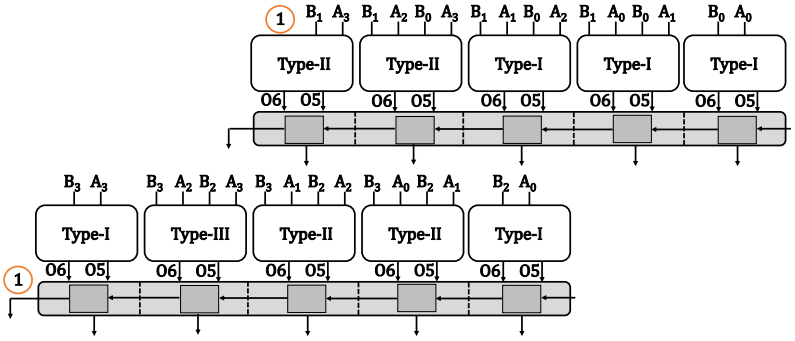


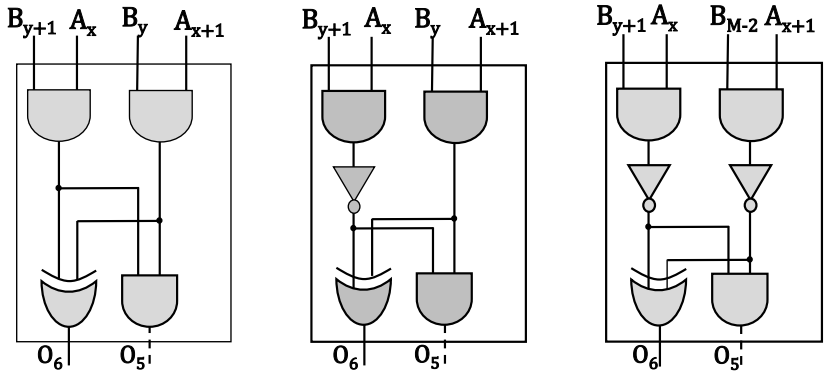
Fig. 5. LUT mapping of partial products for 4×4 accurate signed multiplier

3.1 Accurate Multiplier Design

We have used Baugh-Wooley's multiplication algorithm [4] in this work to implement a signed multiplier. However, the proposed methodology for approximation is equally applicable to other multiplication algorithms, such as Booth's algorithm used in [36]. Baugh-Wooley's multiplication algorithm encodes the required sign extension logic in the generated partial products. Equation 1 describes the multiplication algorithm for $A_{N-bit} \times B_{M-bit}$ signed multiplication. To identify the different types of LUTs configurations (INIT values) to encode Equation 1 and present the methodology, we will consider an example of a 4×4 signed multiplier. Figure 4 shows the different partial product terms according to Equation 1 for a 4×4 multiplier. The (1) in the figure shows the addition of 1's at 2^{N-1}, 2^{M-1} and 2^{N+M-1} locations in the partial products. Utilizing the 6-input LUTs, we can generate and add the consecutive partial product terms using a single LUT. For example, $A_1B_0 + A_0B_1$ can be computed by a single LUT and a single cell of the carry chain. To accommodate the different operations in Figure 4, Figure 5 shows the various types of LUT configurations required to implement the 4×4 multiplier. The outputs of the LUTs, O5 and O6, are provided to the carry chains as carry-generate and carry-propagate signals, respectively, to generate intermediate results. These intermediate results, along with the two (1), are added together to compute the final accurate product. The combinational logic implemented by each LUT configuration is described in Figure 6. The work in [35] has also used a similar implementation for implementing an accurate signed multiplier. However, that design utilizes 5 different types of LUT configurations for the multiplier implementation.

3.2 Approximation Methodology

The proposed approximation methodology is based on disabling LUTs (and the corresponding carry chain cells) in the accurate implementation to introduce approximations. For an M × N accurate multiplier, utilizing K LUTs for partial



(a) Type-I (b) Type-II (c) Type-III

Fig. 6. Configurations of LUTs to implement an accurate multiplier

products generation, we have used a K-bit string (referred to as input configuration) to address each LUT. A $'0'$ at any location in the K-bit string represents the disabling of the corresponding LUT. Disabling a LUT means that the LUT will not contribute to producing the intermediate results. For this purpose, the carry-propagate signal[3] (O6) for the corresponding carry chain cell is fixed to constant $'1'$. Since a disabled LUT will not contribute to intermediate results' value, the carry-generate signal is provided by the external bypass signal, and it is fixed to constant $'0'$. These settings enable the respective carry-chain cell to forward the *preceding carry* without generating a new one. Further, the output of the corresponding carry chain cell (output of XOR) gate is truncated to $'0'$. These settings allow the synthesis tools not to use a disabled LUT during the synthesis and implementation process. For example, Figure 7 shows the proposed approximation methodology for the 4×4 multiplier for a 10-bit configuration string $'1011011001'$. The LSBs $'11001'$ addresses the LUTs in the upper row, and the MSBs $'10110'$ identifies LUTs in the bottom row. For all the 0s in the binary string, the corresponding LUTs have been disabled. For example, for the least significant position in the second row, the O6 signal is fixed to $'1'$; therefore, the respective carry chain cell forwards the carry-in to the next carry chain cell. A bypass signal with the value $'0'$ is used instead of the O5 signal to help the synthesis tool not use the respective LUT. Further, the respective carry chain cell's output is also truncated to $'0'$ to show that the corresponding cell has been disabled. Since an accurate 4×4 multiplier utilizes 10 LUTs for the partial product generation, therefore, the proposed approximation methodology supports 2^{10} approximate multipliers. Similarly, for an accurate 8×8 multiplier utilizing 36-LUTs for partial product generation, *AppAxO*'s proposed approximation methodology provides 2^{36} different approximate multipliers with different accuracy and implementation performance metrics. It should be noted that the proposed approximation methodology is automated, generic and scalable. Therefore, it can be utilized for implementing approximate circuits for any arithmetic operator (of arbitrary bit-width) that utilizes LUTs and carry chains for its implementation.

3.3 Approximate Adders

The utilization of the carry chains in FPGA logic slices facilitates the implementation of an N-bit accurate adder using only N LUTs. Figure 8(a) represents the LUT mapping of an accurate unsigned 4-bit adder. For this purpose, the LUTs compute the required carry-generate (O5) and carry-propagate (O6) signals from the corresponding two input bits

[3]Carry-propagate and carry-generate signals are shown in Figure 3.

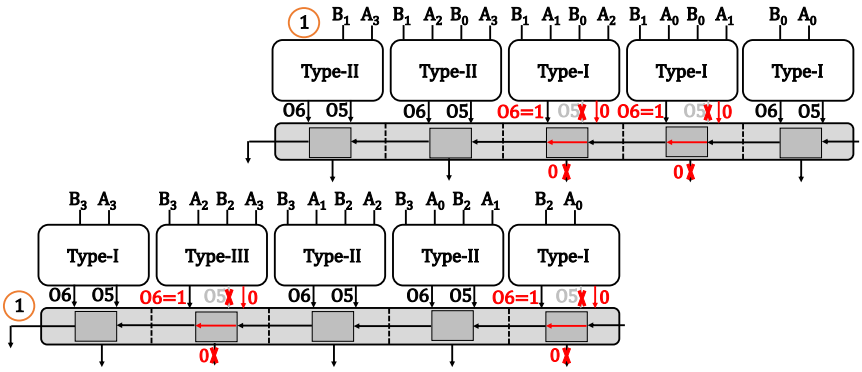
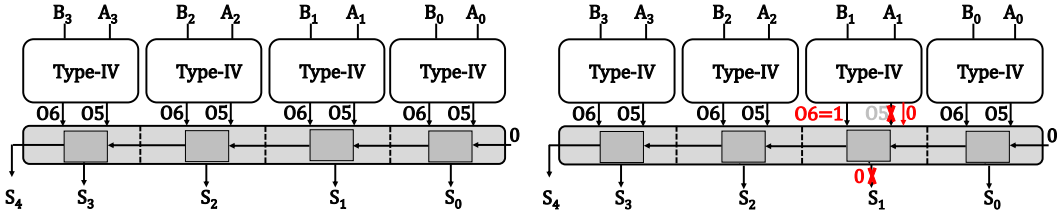


Fig. 7. 4 × 4 approximate multiplier design according to input configuration '1011011001'



(a) Accurate Adder (b) Approximate Adder

Fig. 8. LUT mapping of 4-bit adder. (a) Accurate implementation, (b) Approximate adder design according to input configuration '1101'.

by performing logical *'AND'* and *'XOR'* operations on the input bits, respectively. The LUT configuration Type-IV performs these operations in Figure 8(a). Further, to accommodate the overflows caused by adding two N-bit numbers, we produce an $(N+1)$-bit output sum as shown in the figure. Utilizing the proposed *AppAxO* methodology, we use an N-bit string to address each LUT in the accurate implementation. A $'0'$ value at any position in the N-bit string (input configuration) denotes the disabling of the corresponding LUT and truncating the corresponding output generated by the respective carry chain cell to $'0'$. For instance, Figure 8(b) represents an approximate 4-bit adder implementation for input configuration $'1101'$. As discussed previously, to disable the LUT at the second least significant location, we utilize the external bypass signal to provide a $'0'$ as the carry-generate signal, and O6 provides a constant $'1'$ as the carry-propagate signal. For an N-bit accurate adder, *AppAxO* provides 2^N approximate adders with different accuracy and implementation performance metrics.

4 DSE FOR FPGA-BASED APPROXIMATE OPERATORS SYNTHESIS

4.1 Problem Statement

We can represent the implementation of any arbitrary arithmetic operator by the ordered tuple $O_i(l_0, l_1, ..., l_l, ..., l_{L-1}), \forall l_l \in \{0, 1\}$. The term l_l represents whether the LUT corresponding to the operator's accurate implementation is being used or not and L represents the total number of LUTs of the accurate implementation that may be removed to implement approximation. So, the accurate implementation can be represented as $O_{Ac}(1, 1, ..., 1)$. Similarly, $O = \{O_i\}$ represents the set of all possible implementations of the operator. We can abstract an arbitrary application's behavior by a function $\mathcal{S}$.

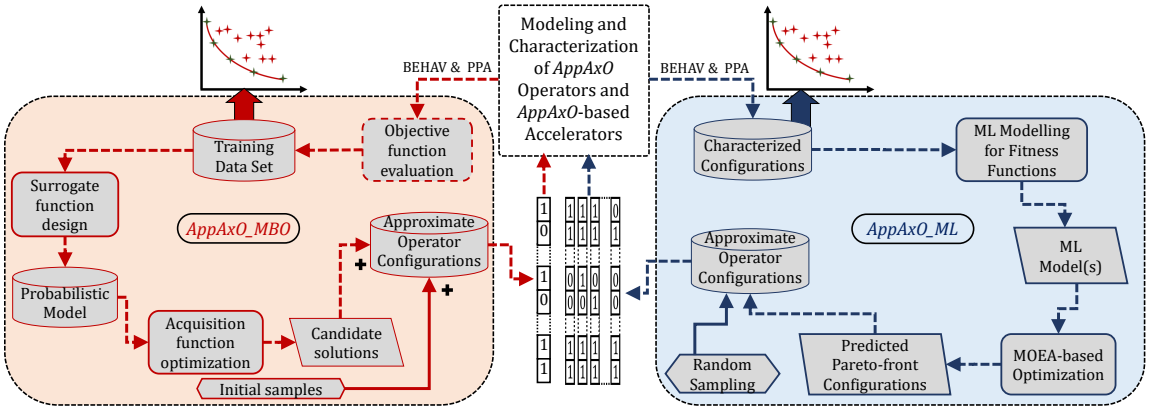


Fig. 9. Proposed design space exploration methods for *AppAxO*

So the output of the application for a set of inputs can be abstracted as shown in Equation 2. The term Err_{O_i} represents the error in the applications behavior as a result of using an approximate operator O_i compared to using the accurate operator O_{Ac}.

$$\begin{aligned} Out_{O_i} &= \mathcal{S}(O_i, Inputs) \\ Out_{O_{Ac}} &= \mathcal{S}(O_{Ac}, Inputs) \\ Err_{O_i} &= Out_{O_{Ac}} - Out_{O_i} \end{aligned} \tag{2}$$

Similarly, the accelerator's hardware performance can be abstracted as a set of functions as shown in Equation 3. Given this framework, the resulting optimization objective can be expressed as shown in Equation 4. Since we are concerned with multi-objective optimization, we obtain a set of non-dominated design points, O_i, that form the Pareto front. Therefore, the solution can be represented as a set $\mathcal{PF}_O = \{O_0, O_1, ...O_{P-1}\}$ representing P number of operator design configurations in the Pareto front.

$$\begin{aligned} Power\ Dissipation: \quad \mathcal{W}_{O_i} &= \mathcal{H}_W(O_i, Inputs) \\ LUT\ Utilization: \quad \mathcal{R}_{O_i} &= \mathcal{H}_R(O_i) \\ Critical\ Path\ Delay: \quad \mathcal{C}_{O_i} &= \mathcal{H}_C(O_i) \\ PDPLUT_{O_i} &= \mathcal{W}_{O_i} \times \mathcal{R}_{O_i} \times \mathcal{C}_{O_i} \end{aligned} \tag{3}$$

$$\underset{O_i \in O}{\text{minimize}}(Err_{O_i}, PDPLUT_{O_i}) \tag{4}$$

The proposed approximation methodology presents an opportunity for the designers to implement operators with improved power, performance and area (PPA) if the application can tolerate the corresponding error profile of the operator. As described in the previous section, for an N-bit adder occupying only N LUTs, *AppAxO* provides 2^N different approximate adders. For example, for 4-bit, 8-bit and 12-bit adders, *AppAxO* provides 2^4, 2^8, and 2^{12} approximate adder designs, respectively. Therefore, all approximate adders can be synthesized and implemented for an application due to adders' comparatively smaller design space. However, it can be noted that the proposed approximation methodology

presents the designer with an exponentially increasing (with the bit-width of multiplier) number of approximate multiplier choices to implement. For instance, the possible number of input configurations of the approximate multiplier increases from 1024 to nearly 64 billion as we consider 8-bit multipliers instead of 4-bits. In order to search for configurations that are appropriate for a given application, the designer can deploy a random search or generate configurations from intuition. However, such approaches do not provide a generalized methodology for designing application-specific approximate operators. Hence, to aid the designer in implementing the appropriate approximate multiplier design, we present two DSE methods—*AppAxO_MBO* and *AppAxO_ML*—as shown in Figure 9. Both of the proposed methods use the automated modeling and estimation methodology described in the previous section.

4.2 DSE using Bayesian Optimization

The left half of Figure 9 shows the various stages involved in Multi-objective Bayesian Optimization (MBO)-based approach to finding the set of Pareto-front approximate operators — both for stand-alone operator designs as well as the operator being used for any specific application. Contrary to randomized algorithm-based optimization methods such as Simulated Annealing and GA, Bayesian optimization allows a more directed search that is particularly useful in case of problems involving costly fitness function evaluations. As shown in Figure 9, the process involves three main stages. Firstly, few randomly sampled initial configurations of approximate operators are generated. The *Objective Function* evaluation involves true characterization of the corresponding operator, and/or the accelerator and the behavioral accuracy of the application using the operator to obtain the design objectives for the initial samples. So, the Objective function can be viewed as the equivalent of $\mathcal{S}$, $\mathcal{H}_W$, $\mathcal{H}_R$ and $\mathcal{H}_C$ from Equation 2 and Equation 3. This set of initial tuples: input configuration $\rightarrow$ design fitness, forms the training data for the second stage, where the *Surrogate Function* generates a probabilistic prediction model. The probabilistic models are approximations of $\mathcal{S}$, $\mathcal{H}_W$, $\mathcal{H}_R$ and $\mathcal{H}_C$ and their estimation accuracy should ideally improve with each iteration. In the third stage, the prediction model(s) is used by the *Acquisition Function* to generate a set of candidate input configurations for the next iteration. As shown by the dotted lines in Figure 9, each iteration involves these three stages with a fixed number of new input configurations undergoing true characterization in each iteration.

It must be noted that MBO is an increasingly frequently used search method. With *AppAxO_MBO* we present a multi-objective optimization method that uses the input configurations of the approximate operator as the design variables for a bi-objective optimization problem. We use separate probabilistic models for each design objective—one that quantifies the application/operator's behavioral accuracy and the second quantifying the PPA design intent. The implementation of the acquisition function involved generating a set of random input configurations, using the surrogate function to predict their fitness, then ranking them according to their expected contribution to the Pareto-front hypervolume and selecting a fixed number of top-ranked samples for the next iteration.

4.3 MOEA-based optimization

In addition to AppAxO_MBO, we present a Multi-Objective Evolutionary Algorithms (MOEA)-based approach to the DSE problem of *AppAxO*. Specifically, we use GA for the bi-objective optimization problem. MOEA-based optimization methods usually rely on low-cost fitness function evaluation and hence, a naive approach to implementing GA would limit the usability in *AppAxO* to small designs with low synthesis time-cost. To this end, as shown in the right half of Figure 9, we present AppAxO_ML, an ML-based DSE approach. We use an initial set of random samples to train multiple ML models for predicting PPA and accuracy metrics of an application. The ML-based models are in turn used during GA-based search to predict the Pareto-front design points, the Predicted Pareto Front (PPF). The collection of

PPFs from each ML model are then evaluated and filtered to provide the Evaluated Pareto Front (EPF). The details of the ML models used in AppAxO_ML are described in the next section. The encoding for GA involves:

- *Individual*: Each input configuration for the approximate multiplier forms an individual in the population.
- *Crossover*: We use *two-point* crossover for exchanging the configuration data of two randomly selected approximate multiplier configurations.
- *Mutation*: *Single-point* mutation was used for randomly altering the configuration of a randomly selected configuration.
- *Selection*: We use a tournament selection method for choosing the configurations to be evaluated in the next generation. We use a tournament size of 3 while selecting the best among those configurations for the next generation.

4.4 ML for DSE

The true evaluation of the application's behavioral accuracy ($\mathcal{S}$) and the accelerator's PPA metrics ($\mathcal{H}_W$, $\mathcal{H}_R$, and $\mathcal{H}_C$) can be a bottleneck in the randomized algorithms-based DSE methods for large design spaces. For instance, the synthesis of the 8-bit accelerator for 2D-convolution of an image of size 128×128 can take up to 20 minutes on a standard computer[4]. To this end, AppAxO utilizes ML-based prediction in the fitness evaluation of AppAxO_ML. In this technique, we use the configuration vector that defines the position of the LUTs used in the approximate multiplier as the input to the ML models. We use a separate model for each behavioral accuracy and PPA metric. The ML models are then used to predict the accuracy, power, CPD, and LUT utilization for any arbitrary configuration. To evaluate the efficacy of our models, we have used the fidelity metric. The fidelity metric denotes the relationship (=, <, >) between the input configurations and their corresponding actual and predicted output values. The lower the Fidelity Error (FE), the higher the correspondence between the predicted metric value the actual value. To further assess our models' efficiency, we also used the Mean Square Error (MSE) and Mean Absolute Error (MAE) of the models' predictions. Equation 5 describe the computation of MSE and MAE values, respectively.

$$\begin{aligned} MSE &= 1/N(\sum_{i=1}^{N}(Y_{pred} - Y_{true})^2) \\ MAE &= 1/N(\sum_{i=1}^{N}|Y_{pred} - Y_{true}|) \end{aligned} \tag{5}$$

The models listed below were used with their most widely used configurations. For training and testing our models, we have used four datasets[5] of 2060 configurations each. Each configuration shows the impact of a unique approximate multiplier[6] on the output accuracy and the corresponding performance metrics. Our model employs randomly chosen 80% configurations of the input dataset for training the model. The remaining 20% of the input dataset is used for testing the trained model.

(1) Random Forest Regression (RFR): It is an ensemble learning-based supervised algorithm. The ensemble technique utilizes multiple ML models for predictions. These predictions are then combined to make more accurate predictions

[4]For 8-bit arithmetic, our methodology can provide up to 2^{36} approximate signed multipliers.
[5]One for multipliers and one for each application referred to in Figure 1
[6]8×8 multipliers use a 36-bit string to represent LUTs for partial products generation as described in Section 3.2.

Table 2. The ML models reporting the minimum error in terms of Mean Square Error (MSE), Mean Absolute Error (MAE) and Fidelity Error (in %) for the three applications. ML models used: Random Forest Regression (RFR), Support Vector Regression (SVR), Stochastic Gradient Descent (SGD), Gradient Boosting Regression (GBR), Decision Tree Regression (DTR), Multilayer Perceptron (MLP)

Metric	Models with minimum MSE (Train, Test)			Models with minimum MAE (Train, Test)			Models with minimum Fidelity Error (Train, Test)		
Application	ECG	MNIST	GS	ECG	MNIST	GS	ECG	MNIST	GS
App-specific Accuracy	RFR, RFR	RFR, MLP	RFR, MLP	RFR, RFR	RFR, RFR	RFR, GBR	RFR, RFR	RFR, RFR	RFR, GBR
CPD (in nS)	RFR, RFR	RFR, RFR	RFR, RFR	RFR, RFR	RFR, RFR	RFR, GBR	RFR, DTR	RFR, GBR	RFR, SGD
Power (in uW)	RFR, GBR	RFR, GBR	SGD, SGD	RFR, GBR	RFR, DTR	SGD, SGD	RFR, GBR	SVR, RFR	RFR, SGD
LUT Utilization	SGD, MLP	SGD, SGD	SGD, SGD	MLP, MLP	SGD, SGD	SGD, SGD	RFR, MLP	RFR, SGD	RFR, SGD

than a single ML model. It operates by constructing a multitude of decision trees at training time [18]. Each tree draws a random sample from the original data set when generating its splits, adding an additional randomness element that prevents overfitting of the model.

(2) Support Vector Regression (SVR): Compared to a simple linear regression, SVR allows the specification of an acceptable error margin and tolerance of a model for the specified error margin [31]. However, SVR requires specific calibrations of different features, which makes it less robust.

(3) Stochastic Gradient Descent (SGD): It is an optimization technique and is generally used in sparse ML problems. The gradient of the loss is estimated (each sample at a time), and the model is updated accordingly with a decreasing learning rate [6]. It is, however, very sensitive to feature scaling and requires tuning of several hyperparameters.

(4) Gradient Boosting Regression (GBR): In this technique, a predictive model is produced from an ensemble of weak predictive models [11]. It is similar to RFR; however, the individual predictive models are combined at the start to produce an output, whereas it is done at the end for RFR.

(5) Decision Tree Regression (DTR): These are a series of sequential steps designed to solve a problem and provide probabilities, costs, or other consequences of making a particular decision. Smaller subsets are broken down from a dataset while the associated decision tree is incrementally developed at the same time [3].

(6) Multilayer Perceptron (MLP): It is a class of feedforward ANN and uses backpropagation for training. MLP learns a function between the input features and the output by using intermediate hidden layers [24].

The results of ML models giving the minimum error for each of the metrics across different applications have been tabulated in Table 2. The processing time needed for the training and the average inference time for each configuration is shown in Table 3. The inference time is averaged over the total time consumed in performing inference for the complete dataset.

5 EXPERIMENTS AND RESULTS

5.1 Experiment Setup

We have implemented all presented multipliers in VHDL and synthesized them for the $7VX330T$ device of the Virtex-7 family (unless stated otherwise) using Xilinx Vivado 19.2. Our methodology implements each operator—adder and multiplier—design multiple times to obtain precise critical path delay and dynamic power consumption values. According to the previous iteration's critical path-slack, our tool flow modifies the new critical path delay constraint in each iteration

Table 3. The execution time for training and inference on various ML models for different target metrics. All the timing values are reported in *milliseconds*. The inference timing values are reported for a single data point averaged over the inference of the whole training set (2060 points). ML models used: Random Forest Regression (RFR), Support Vector Regression (SVR), Stochastic Gradient Descent (SGD), Gradient Boosting Regression (GBR), Multilayer Perceptron (MLP)

Application	Target Metric for ML Model	GBR		MLP		RFR		SGD		SVR	
		TRAIN	INFER	TRAIN	INFER	TRAIN	INFER	TRAIN	INFER	TRAIN	INFER
ECG	Accuracy	2023.721	0.377	242.924	0.175	698.487	6.05	4.185	0.097	14.323	0.103
	CPD	2060.872	0.403	708.342	0.244	751.05	7.214	6.456	0.097	225.132	0.179
	Power	2131.917	1.215	4505.54	0.179	1010.047	8.782	24.82	0.096	241.39	0.219
	LUTs	2154.619	0.403	3673.557	0.179	678.572	6.056	34.46	0.105	245.992	0.237
GS	Accuracy	2004.419	0.334	2857.172	0.177	706.297	5.789	12.686	0.096	262.942	0.186
	CPD	2331.065	0.422	2614.81	0.186	706.558	6.391	33.315	0.128	221.338	0.19
	Power	2259.041	0.463	4592.353	0.197	712.243	6.289	55.562	0.097	250.689	0.248
	LUTs	2130.301	0.469	5095.321	0.198	708.136	5.961	40.898	0.105	237.082	0.219
MNIST	Accuracy	2033.754	0.42	4296.751	0.171	692.762	5.878	36.383	0.093	264.139	0.187
	CPD	2127.168	0.588	2581.044	0.197	702.273	6.042	29.538	0.099	274.11	0.203
	Power	2397.082	0.51	5140.923	0.186	925.009	7.447	116.668	0.121	272.108	0.22
	LUTs	1993.787	0.342	4202.479	0.174	762.154	5.865	46.64	0.094	243.259	0.196
MULT 8 X 8	Average Absolute Error	2158.514	0.329	4462.492	0.182	792.243	6.247	34.367	0.103	235.551	0.201
	Average Absolute Relative Error	2014.313	0.343	3820.165	0.169	689.845	6.253	31.939	0.098	258.572	0.215
	CPD	2883.947	0.393	889.988	0.176	769.147	8.08	28.563	0.215	205.164	0.25
	Power	2038.58	0.456	3105.816	0.178	693.654	5.665	9.651	0.1	232.453	0.2
	LUTs	2185.196	0.51	4232.282	0.207	685.203	6.134	10.355	0.097	262.186	0.197

of the implementation. Using this characterization method, the accuracy and PPA estimation for each approximate 8×8 multiplier configuration consumes nearly 3.55 minutes of processing time. The accelerators for the applications were implemented using different high-level languages. For the calculation of the dynamic power of all implementations, Vivado Simulator and Power Analyzer tools have been utilized. All applications have been implemented for Xilinx Zynq UltraScale+ MPSoC (xczu3eg-sbva484-1-e device). All behavioral estimations were implemented in Python. The ML-based modeling, and the MBO-based DSE methodology were implemented in Python using multiple packages, including scikit-learn, TensorFlow [1], PyGMO [5]. All experiments were conducted on an HPC server with a single AMD EPIC^{TM} processor with 24 cores and two PCIe Gen4 NVIDIA A100 Tensor-Core-GPUs, with 512GB of DDR4-3200MHz main memory. In our current work, we focus on hardware performance as a result of using approximate operators, and the analysis and mitigation of precision scaling-induced errors for any application are beyond the scope of this work. The following applications were used in the experiments for demonstrating the effectiveness of *AppAxO*. It must be noted that we have used homogeneous accelerator designs for each application. Therefore, all the approximate multiplier designs used in any arbitrary accelerator use similar LUT configurations.

5.1.1 ECG Peak Detection. To represent the set of applications using one-dimensional convolution, we use the peak detection of ECG signal on Physionet's single-lead ECG dataset [8] using Pan–Tompkins algorithm [25] as the test application. Pan-Tompkins algorithm consists of 5 stages – *Low Pass Filtering, High Pass Filtering, Derivative Filtering, Squaring, Moving Window* and *Peak Finder*. For the current work, we explored the effect of approximate multipliers in the low pass filter. We use *accuracy*, evaluated as the ratio of the *True Positives* and the sum of *True Positives, False Negatives* and *False Positives* of the detected peaks, as the relevant behavioral metric for our current work. The behavioral estimation for a single test case of ECG peak detection using 8×8 approximate multiplier configuration expends

an average of 1.5 minutes of processing time. Similarly, the accelerator synthesis and implementation consume 1.73 minutes of processing time.

5.1.2 Gaussian Smoothing. We use Gaussian Smoothing (GS), a frequently used benchmark for representing the set of 2D convolution-based applications. In *AppAxO*, we implemented a 5×5 kernel, and the accelerator, implemented using Vivado HLS, uses a line buffer, along with 25 multipliers. We use the average reduction in the Peak Signal-to-Noise Ratio (PSNR) as the behavioral minimization objective. It represents the negative of the PSNR improvement using GS with an approximate multiplier over 45 images. The behavioral estimation for a single test case of GS using 8×8 approximate multiplier configuration expends an average of 3.35 minutes of processing time. Similarly, the accelerator synthesis and implementation consume 5.06 minutes of processing time.

5.1.3 MNIST digit Recognition. Image classification is a commonly used application for evaluating the efficacy of various approximation techniques. We have used a lightweight MLP implemented in Python for the classification of the MNIST Digit dataset [2, 9]. The MLP consists of two fully connected hidden layers having 100 and 32 nodes, respectively. For this work, we have evaluated the efficacy of the various approximate multipliers by deploying them in the output layer of the network for inference using $10,000$ test images. For this purpose, we have quantized the floating-point trained weights and input activations of the last layer according to the multiplier size under consideration. To compute the application's performance metrics, we have implemented the last layer in Vivado HLS and evaluated for different approximate multipliers. We shall refer to this application as MNIST in the discussion of the experiment results. The behavioral estimation for a single test-case of MNIST using 8×8 approximate multiplier configuration expends an average of 0.78 minutes of processing time. Similarly, the accelerator synthesis and implementation consumes 2.52 minutes processing time.

The approximation-aware DSE for both operator-level and application-level design involves finding multiple design points that provide varying levels of behavioral (accuracy) and PPA trade-offs. For our current work, we use application-specific accuracy metric and the product of PDP and LUT utilization as the two objectives. Consequently, the DSE runs for the experiments involve multi-objective optimization (with two objectives). Therefore, we use hypervolume of the Pareto-front and the number of non-dominated design points, two commonly used metrics for comparing the results from different multi-objective optimization runs. The hypervolume indicator measures the significance of the non-dominated design points by computing the volume of the dominated portion of the objective space. For a two-objective problem, the hypervolume corresponds to the area between the non-dominated Pareto-front and a *reference point*. For our current work, we aim at minimizing both the approximation induced error metric and PDP x LUT. Hence, the reference point comprises of the maximum of both the metrics across all Pareto-front points under consideration.

5.2 Accuracy-Performance Analysis of Approximate Adders

For an adder utilizing N LUTs and corresponding carry chain elements, our proposed approximation methodology generates 2^N approximate adder designs. The adder having input configuration $2^N - 1$ utilizes all LUTs, and it is an accurate adder. For example, for a 4−bit and 12−bit adder, our methodology generates 2^4 and 2^{12} different approximate adders, respectively. For example, Figure 10 compares the hardware performance metrics and accuracy of the *AppAxO*-generated fifteen 4-bit approximate adders. Please note that we do not synthesize hardware for configuration $'0000'$ (all LUTs disabled). For assessing the performance of the designs, we have used the PDP $\times$ LUT metric to incorporate all

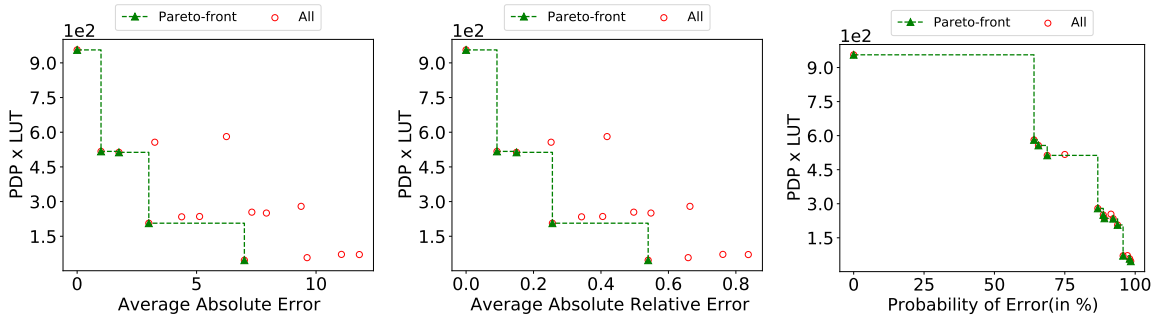


Fig. 10. Accuracy and performance comparison of *AppAxO*-generated 4-bit unsigned adders. The power and CPD are in μW and ns, respectively.

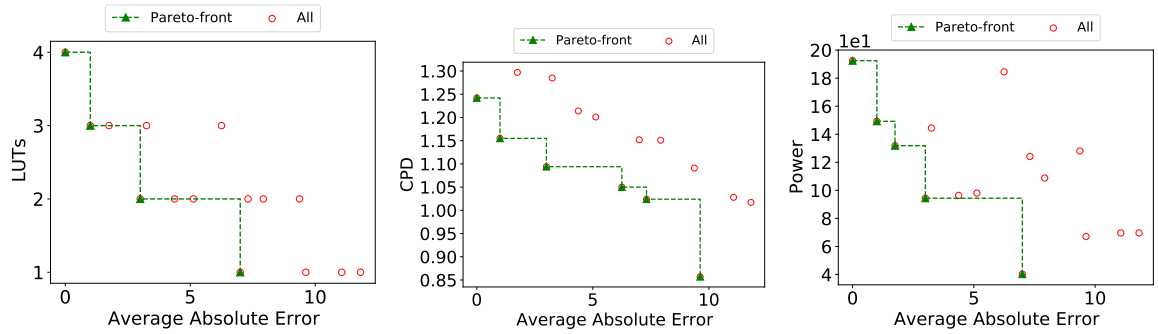
design metrics. A smaller value of $PDP \times LUT$ represents a circuit with better performance. Similarly, we have used three different error metrics for accuracy evaluation of the individual adders, i.e., Average Absolute Error, Average Absolute Relative Error, and Error Probability. Equation 6, Equation 7, and Equation 8 define these error metrics, respectively. Figure 10(a) shows that a total of 5 design configurations ('1111', '1110', '1101', '1100', and '1000') lie on the Pareto front. The non-dominated adder configuration '1111' utilizes all four LUTs and is an accurate adder. Moreover, all non-dominated design points have the most significant LUT enabled. An interesting design point is the adder configuration '1000', which utilizes only a single LUT. Figure 10(b) compares the average absolute relative error and $PDP \times LUT$ of the fifteen design points. The analysis returns the same five non-dominated designs as described in Figure 10(a). The analysis in Figure 10(c) shows a total of 12 non-dominated design points. These points also include four non-dominated design points from Figure 10(a).

$$Average\ Absolute\ Error = \frac{\sum_{\forall inputs} |Accurate_{result} - Approx_{result}|}{Total\ number\ of\ inputs} \tag{6}$$

$$Average\ Absolute\ Relative\ Error = \frac{\sum_{\forall inputs} |\frac{Accurate_{result} - Approx_{result}}{Accurate_{result}}|}{Total\ number\ of\ inputs} \tag{7}$$

$$Error\ Probability = \frac{Total\ number\ of\ inputs\ for\ which\ Approx_{result} \neq Accurate_{result}}{Total\ number\ of\ inputs} \times 100 \tag{8}$$

Figure 11 further elaborates the trade-offs between accuracy and individual performance metrics of the fifteen 4-bit approximate adders. Figure 11(a) has four non-dominated design points with 4, 3, 2, and 1 utilized LUT(s), respectively. These design points correspond to configurations '1111',' 1110',' 1100', and '1000' and they produce average absolute error values of 0, 1, 3, and 7, respectively. All of these designs are also part of the non-dominated design points in Figure 10(a). Figure 11(b) shows six non-dominated design points, including the configurations '1111',' 1110', and '1100'. Similarly, the non-dominated design points in Figure 11(c) also include configurations '1111',' 1110',' 1100', and '1000'. The non-dominated design point '1101' in Figure 10(a) is also a non-dominated design point in the error-power analysis in Figure 11(c).



(a) (b) (c)

Fig. 11. Comparing hardware performance and Average absolute error of *AppAxO*-generated 4-bit unsigned adders. (a) LUT utilization (b) Critical path delay (in ns) (c) Power dissipation (in μW)

5.3 Accuracy-Performance Analysis of Approximate Multipliers

For a multiplier utilizing K LUTs for partial product generation, our proposed approximation methodology generates 2^K different approximate multipliers. The multiplier having input configuration $2^K - 1$ utilizes all LUTs, and it is an accurate multiplier . For example, for a 4×4 and an 8×8 multiplier, our methodology generates 2^{10} and 2^{36} different approximate multipliers, respectively.

5.3.1 Multiplier-level Analysis: To evaluate the efficacy of the generated multipliers, Figure 12 presents an accuracy-performance analysis of all the 4×4 generated multipliers. For the analysis, we have excluded multiplier configuration $'0000000000'$, which disables all the LUTs in the partial product generation stage[7]. For evaluating the performance of the multipliers, we have used the PDP $\times$ LUT metric. Figure 12(a) analyzes the average absolute error and PDP $\times$ LUT metric of all 1023 multipliers. The analysis shows that a total of 38 multiplier designs lie on the Pareto front. The non-dominated multiplier configuration $'1023'$ (binary value $'1111111111'$) utilizes all ten LUTs for partial products generation and produces 0 average absolute error. Most of the non-dominated design points have LUTs enabled at most significant locations. An interesting observation is the non-dominated multiplier designs that enable only a single LUT for partial product generation. For example, non-dominated multiplier designs with configurations 2, 4, 8, 1, and 512 enable only a single LUT for their partial products. Designs 1 and 512 have the binary configuration $'0000000001'$ and $'1000000000'$ respectively. Design 1 utilizes only the least significant LUT in the first partial product row, and design 512 deploys only the most significant LUT in the second row of partial products. Design 512 has a lower average absolute error and higher PDP $\times$ LUT than design 1.

Figure 12(b) presents the Pareto analysis of the 1023 design points for PDP $\times$ LUT and average absolute relative error metrics. Compared to the non-dominated designs in Figure 12(a), average absolute relative error introduces five new multipliers to the set of Pareto designs. Similarly, the comparison in Figure 12(c) shows a total of 20 non-dominated multipliers out of 1023 designs. These points also include 11 new multiplier designs, which were dominated in the other two plots. However, the generated approximate multipliers' application-specific efficacy cannot be determined from the accuracy-performance analysis of individual multipliers. For this purpose, either the generated approximate multipliers

[7]The mapping of multiplier configuration (a binary string) to LUTs is described in Figure 7.

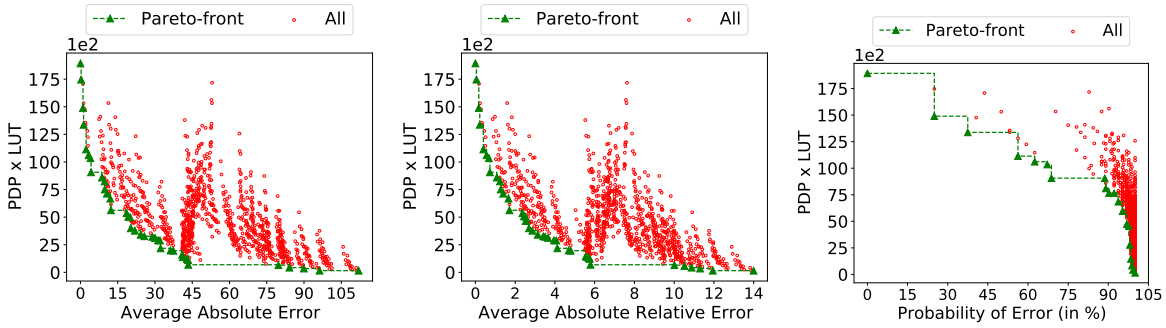


Fig. 12. Accuracy-performance analysis of *AppAxO* generated 4 × 4 approximate multipliers. The power and CPD are in μW and ns, respectively.

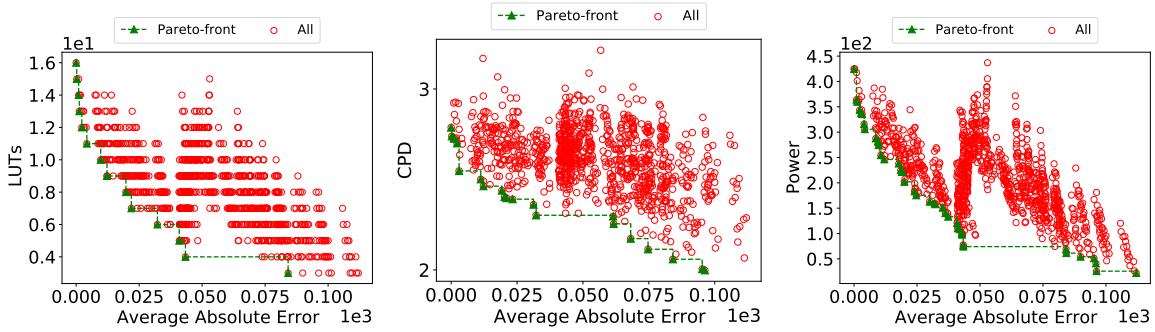
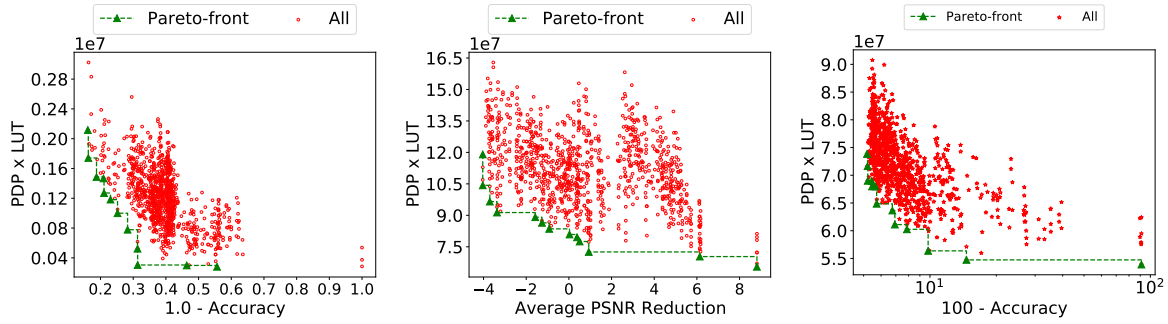


Fig. 13. Comparing hardware performance and Average absolute error of *AppAxO*-generated 4 × 4 signed multipliers. (a) LUT utilization (b) Critical path delay (in ns) (c) Power dissipation (in μW)

should be exhaustively utilized in an application's implementation, or some machine learning-based intelligent models should be used to estimate the potential efficiency of the various multipliers.

Figure 13 further elaborates the trade-offs between accuracy and individual performance metrics of the 1023 4 × 4 approximate multipliers. These results also include the implementation of a binary adder to add the generated partial products to compute the final product. For example, the non-dominated design configuration '1111111111' in Figure 13 employs 10 LUTs for partial products generation and 6 LUTs for the binary adder to utilize a total of 16 LUTs to compute the final product. The accuracy-performance analysis in Figure 13 reveals that the non-dominated multiplier configurations producing low average absolute error values have most of the LUTs enabled in the significant partial product row. For example, multiplier configuration '1111111111' is an accurate multiplier (with 0 average absolute error) is a non-dominated design point in all three sub-figures in Figure 13. Similarly, multiplier configuration '1111111110'—least significant LUT in the least significant partial product row disabled—is the non-dominate design point having the second-lowest average absolute error value (0.25) in Figure 13(a) and Figure 13(b). The non-dominated design point having the second-lowest average absolute error in Figure 13(c) is '1111011111'—least significant LUT in the significant



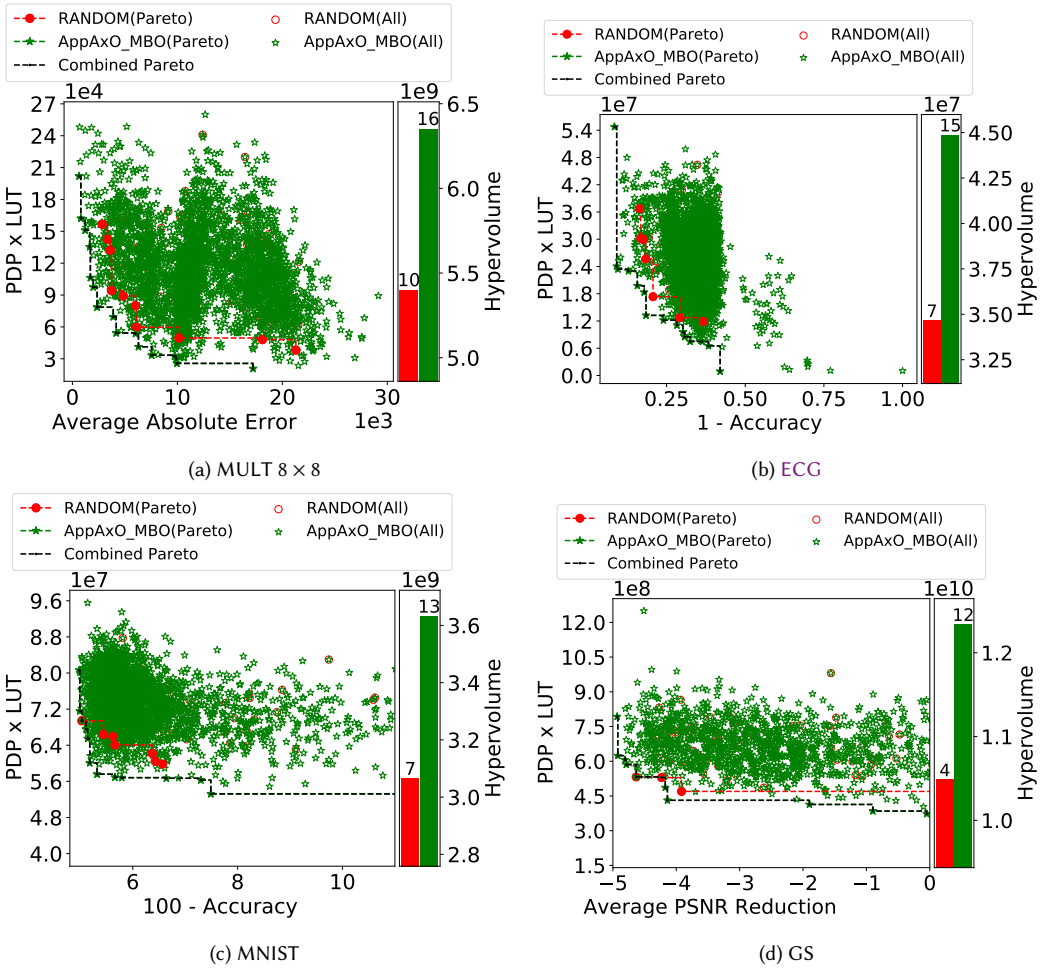
(a) ECG (b) GS (c) MNIST

Fig. 14. Accuracy-performance analysis of approximate 4×4 multipliers in three different applications. The power and CPD are in μW and ns, respectively.

partial product row disabled. Both of these configurations, '1111111110' and '1111011111', are also non-dominated design points in Figure 12(a).

5.3.2 Application-level Analysis of Approximate 4×4 *Multipliers:* Figure 14 shows the utilization of all 4×4 approximate multipliers[8] in three different applications. The output accuracy of each application has been computed by utilizing the multipliers' behavioral models in the high-level implementation of each application. Similarly, the performance metric (PDP $\times$ LUT) has been computed by utilizing the multipliers' VHDL implementation in the accelerator of each application. Figure 14(a) shows the accuracy-performance analysis of the ECG application. A total of 12 different approximate multipliers-based designs are non-dominated design points with different accuracy and performance parameters. Six designs among these non-dominated designs utilize multipliers which were among dominated points in Figure 12. It is interesting to note that the accurate multiplier-based accelerator is not among the non-dominated design points. The ECG application's inherent error-tolerance significantly masks the approximations-generated errors and removes the accurate multiplier-based design from the set of Pareto points. The approximate multiplier with binary configuration '1111011111' (decimal value 1007) produces the highest output accuracy. Moreover, the application-specific accuracy-performance analysis can reveal highly efficient multipliers for that application. For example, approximate multipliers with configuration values 16, 128, and 512 utilize a single LUT for partial product generation; however, their corresponding accelerators are among non-dominated design points. Figure 14(b) and Figure 14(c) show the utilization of the approximate multipliers for the Gaussian Smoothing filter and MNIST dataset classification MLP respectively. For both applications, a total of 13 different approximate multipliers-based designs lie on the Pareto fronts. Further, for both applications, the accurate multiplier-based designs are not among the non-dominated accelerator designs. Similar to the ECG application's error-resilience, the Gaussian Smoothing Filter and the MNIST classification expose multipliers that utilize only a single LUT for partial product generation. For example, for the Gaussian Smoothing filter, approximate multipliers with configuration values 1, 4, and 16 generate non-dominated design points. These application-specific accuracy-performance analyses augment the need for designing approximate multipliers according to application accuracy-performance requirements. We have also evaluated the efficacy of the *AppAxO* methodology for 8×8 multipliers using various ML models. These results are described in the following subsections.

[8]Excluding multiplier configuration with binary value '0000000000'



(a) MULT 8 × 8

(b) ECG

(c) MNIST

(d) GS

Fig. 15. Comparison of DSE results from AppAxO_MBO against that from initial random samples.

5.4 AppAxO_MBO

To show the effectiveness of the MBO-based exploration, we compared the Pareto-front obtained by AppAxO_MBO starting with 100 random samples to that obtained by the initial random samples in the search for 8 × 8 approximate multipliers. The DSE run configuration includes – 200 iterations with 1000 acquisition samples and 10 true evaluations per iteration. It must be noted that the true evaluations include actual synthesis and implementation of the multipliers accelerators along with Python-based behavioral estimation for each configuration. As a result, the processing time for each DSE run consumed nearly a week of processing time. Figure 15 shows the results in the search for stand-alone multipliers and that for the 3 test applications. The bar plots in each sub-figure show the comparison of the hypervolume and the labels on top of each bar shows the number of points in the corresponding Pareto-fronts. As seen on the figure, the MBO-based DSE resulted in improved hypervolume for all cases, with maximum benefits observed for ECG.

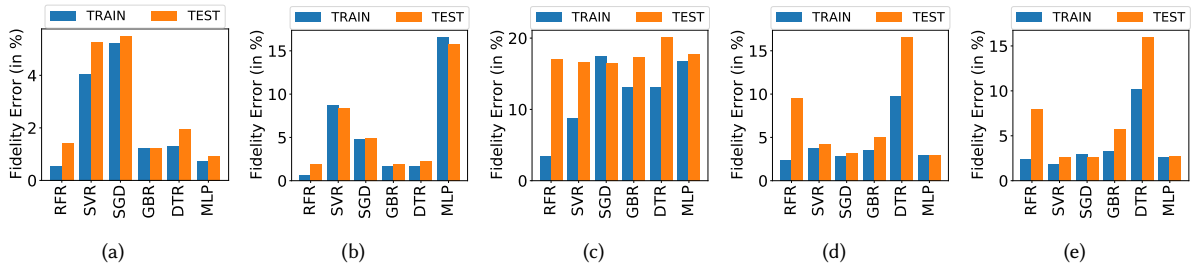


Fig. 16. Fidelity Error in the ML models for different performance metrics of 8 × 8 multipliers. (a) Average Absolute Relative Error (b) Average Absolute Error (c) CPD (in nS) (d) Power (in uW) (e) LUT Utilization

However, the MBO-based search results in a large number of points being synthesized, many of which are not on the Pareto-front.

5.5 ML Modeling

We see the performance of the ML models described in Section 4.4 on the following metrics:

(1) 8 x 8 Multiplier Performance Metrics (Figure 16) - We see that test FE for average absolute relative error metric is lowest for MLP at 0.91%, followed by GBR and RFR. The test MSE and MAE are also lowest for MLP regressor, with each being 0.52 and 0.50, respectively. For CPD, we see the test-FE scores are lowest for SGD at 16.41%, with test-MSE of 0.017 and test-MAE of 0.10 being the lowest for SVR. For Power and LUTs estimations, MLP gives a good performance with test-FE of 2.95%, test MSE of 753.81, and test-MAE of 21.99 for power values. SGD offers the best test-FE of 2.62%, and SVR provides a good test-MSE of 0.81 and test-MAE of 0.71 for LUTs.

(2) ECG Performance Metrics - We see that test Fidelity Error of Accuracy is the lowest for RFR at 3.86%. The test-MSE and test-MAE are also lowest for RFR, with each of them being 0.00041 and 0.01, respectively. For CPD, we see the test-FE is lowest for DTR at 23.23%, test-MSE lowest for RFR of 0.02, and test-MAE for RFR of 0.13. GBR gives low test-FE for Power at 4.56%. The lowest test-MSE of 1585410.54, and the lowest test-MAE of 922.17, are observed for GBR. For LUTs, a good performance is observed with MLP with test-FE of 2.85%, test-MSE as 105.93, and 8.21 as test-MAE.

(3) MNIST Performance Metrics - We see that test Fidelity Error of Accuracy is the lowest for RFR at 6.30%. The test-MSE is lowest for MLP at 2.53, and test-MAE with RFR at 0.71 is the lowest. For CPD, we see the test-FE is lowest for GBR at 20.31%, and test-MSE and test-MAE are lowest for RFR at 0.45 and 0.52, respectively. RFR gives low test-FE for Power at 17.75%. The lowest test-MSE of 12.30 is observed for GBR, and the lowest test-MAE of 2.23 is observed for DTR. For LUTs, a good performance is observed with SGD with test-FE of 2.89%, test-MSE as 1415.29, and 29.9 as test-MAE.

(4) GS Performance Metrics - We see that test FE of Accuracy is the lowest for GBR at 2.94%. The test-MSE is lowest for MLP at 0.25, and test-MAE is lowest for GBR at 0.34. For CPD, we see the test-FE is lowest for SGD at 17.40% with test-MSE of 0.04 with RFR, and test-MAE is lowest for GBR at 0.15. SGD gives low test-FE for Power at 3.17%. The lowest test-MSE of 83177.84 and the lowest test-MAE of 230.61 are observed for SGD. For LUTs, the best performance is observed with SGD with test-FE of 2.89% test-MSE as 601.65 and 19.70 as test-MAE.

Table 4. Processing time (in minutes and seconds) for running GA-based DSE using the ML models for the fitness estimation

Application	PPA	Behavioral	GBR	MLP	RFR	SELECT	SGD	SVR
ECG	PDP x LUT	1 - Accuracy	3m18s	1m27s	55m12s	14m51s	0m43s	1m31s
MNIST	PDP x LUT	100 - Accuracy	3m47s	1m29s	54m23s	27m16s	0m43s	1m42s
GS	PDP x LUT	Average PSNR Reduction	3m22s	1m27s	54m13s	1m24s	0m43s	1m42s
MULT 8×8	PDP x LUT	Average Absolute Error	6m34s	2m19s	94m34s	15m48s	0m56s	2m39s
MULT 8×8	PDP x LUT	Average Absolute Relative Error	5m51s	2m19s	96m6s	15m57s	0m58s	2m41s

Table 5. DSE-level evaluation of ML models. The number of design points in the Predicted Pareto Front (PPF) and Evaluated Pareto Front (EPF) are reported for the DSE runs for six experiments.

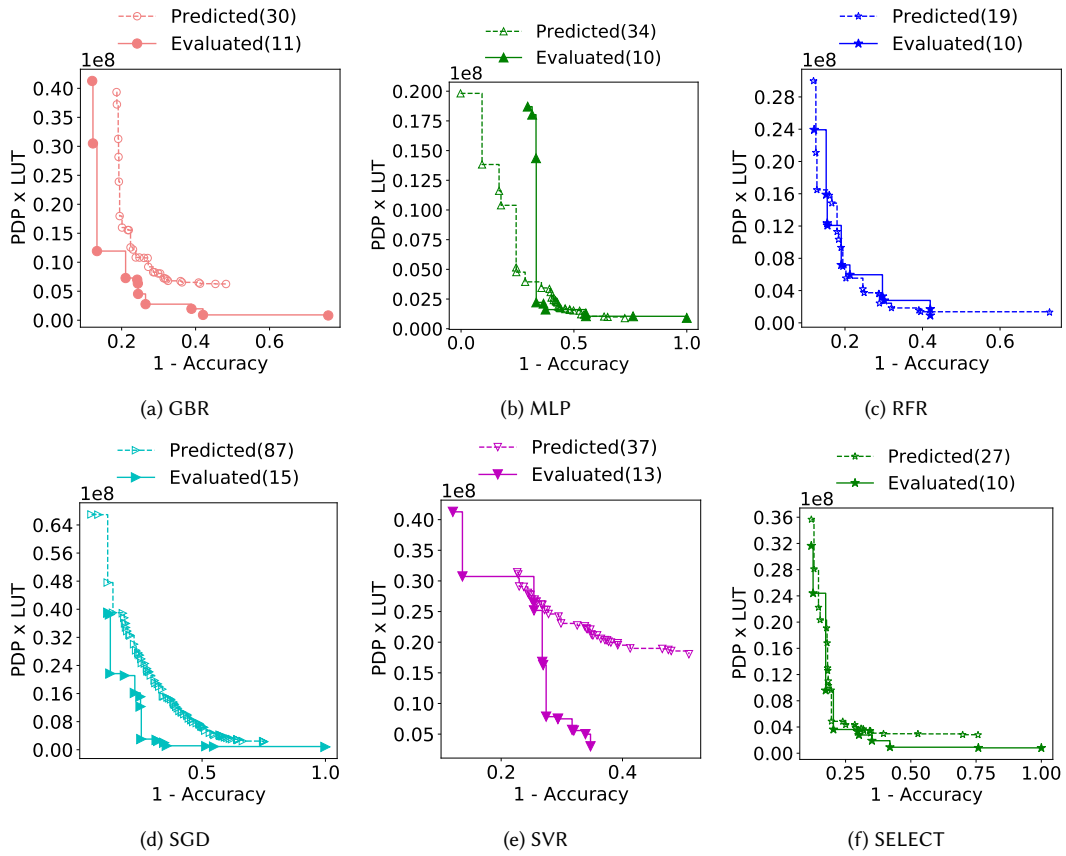
Application	Objectives		GBR		MLP		RFR		SGD		SVR		SELECT	
	PPA	Behavioraal	PPF	EPF	PPF	EPF	PPF	EPF	PPF	EPF	PPF	EPF	PPF	EPF
ECG	PDP x LUT	1 - Accuracy	30	11	34	10	19	10	87	15	37	13	27	10
MNIST	PDP x LUT	100 - Accuracy	39	13	15	9	31	11	43	10	28	8	17	8
GS	PDP x LUT	Average PSNR Reduction	32	10	34	13	39	13	14	10	49	9	17	11
MULT 8×8	PDP x LUT	Average Absolute Relative Error	25	17	29	20	35	22	22	13	32	16	27	21
MULT 8×8	PDP x LUT	Average Absolute Error	28	21	7	4	46	28	30	20	30	15	27	20

Based on the above discussion, it can be concluded that different ML models provide varying levels of accuracy in predicting the performance metrics of an application. Further, as shown in Table 3, the inference time for each model can vary across a wide range. The designer can make a decision, regarding the choice of ML model to use for DSE, based on these two factors – accuracy and processing (inference) time. As we shall show next, collecting the results from DSE runs with varying types of ML models provides the best Pareto-front for each application. However, in a time-constrained scenario, using a selection of ML models (based on their fidelity error performance) can provide results that are close to the results collected from multiple DSE runs using varying models.

5.6 DSE using ML Models

The ML models described in the earlier section were used in the GA-based DSE for 8×8 approximate multipliers. The experiments included six sets of DSE runs. Five experiments involved using each of the ML across every performance metric prediction. The sixth experiment, SELECT, involved choosing the model with the lowest fidelity test error for each performance metric. DTR experiments are not reported since all evaluated points using DTR were reported as Pareto-front points. Table 4 shows the processing time for executing the GA-based DSE using the ML models for fitness evaluation across all the experiments. The GA-specific configurations for the DSE experiment include: 200 generations with starting population of 1000 samples, mutation and cross-over probability of 0.04 and 0.8, respectively, and tournament-based selection with a tournament size of three.

Each of the DSE experiments resulted in a Predicted Pareto Front (PPF), where the metrics correspond to that obtained using ML-based predictions. Then, the set of approximate multiplier configurations from each PPF were evaluated with actual hardware synthesis and behavioral testing, followed by Pareto-front determination with the



(a) GBR (b) MLP (c) RFR

(d) SGD (e) SVR (f) SELECT

Fig. 17. Comparison of the accuracy of the prediction of Pareto-front design points predicted by the ML models for ECG. Models: Gradient Boosting Regression (GBR), Multilayer Perceptron (MLP), Random Forest Regression (RFR), Stochastic Gradient Descent (SGD), Support Vector Regression (SVR), SELECT: Uses the model with the lowest Testing Fidelity Error for each metric.

actual metrics to obtain the Evaluated Pareto Front (EPF). Table 5 shows the number of points in PPF and EPF of each experiment. As expected, not all points in PPF translated to actual Pareto-front design points in the EPF. Figure 17 shows the EPF (Evaluated) and the PPF (Predicted) for the experiments with ECG for all six sets of experiments. RFR and SELECT show the closest match between the PPF and the EPF.

To evaluate the quality of results with each model, we show the EPFs along with a combination of the EPF (All) for the experiments in Figure 18. The numbers in the parenthesis of each label correspond to the contribution of each model to the combined Pareto-front. The hypervolume of all the models, along with the combined results (ALL), is shown in Figure 19, for different application and multiplier DSE experiments. It can be observed that ideally, the combination of results from each of the ML models along with the selective model assignment (SELECT) results in the highest quality of results.

Figure 20 shows the comparison of the Pareto-front of the designs obtained from three different ways. *AppAxO_RND* denotes the set of 2048 randomly generated design configurations for the approximate multiplier. Additionally, 12 configurations denoting corner-case designs were added to *AppAxO_RND* to generate the set of 2060 design points,

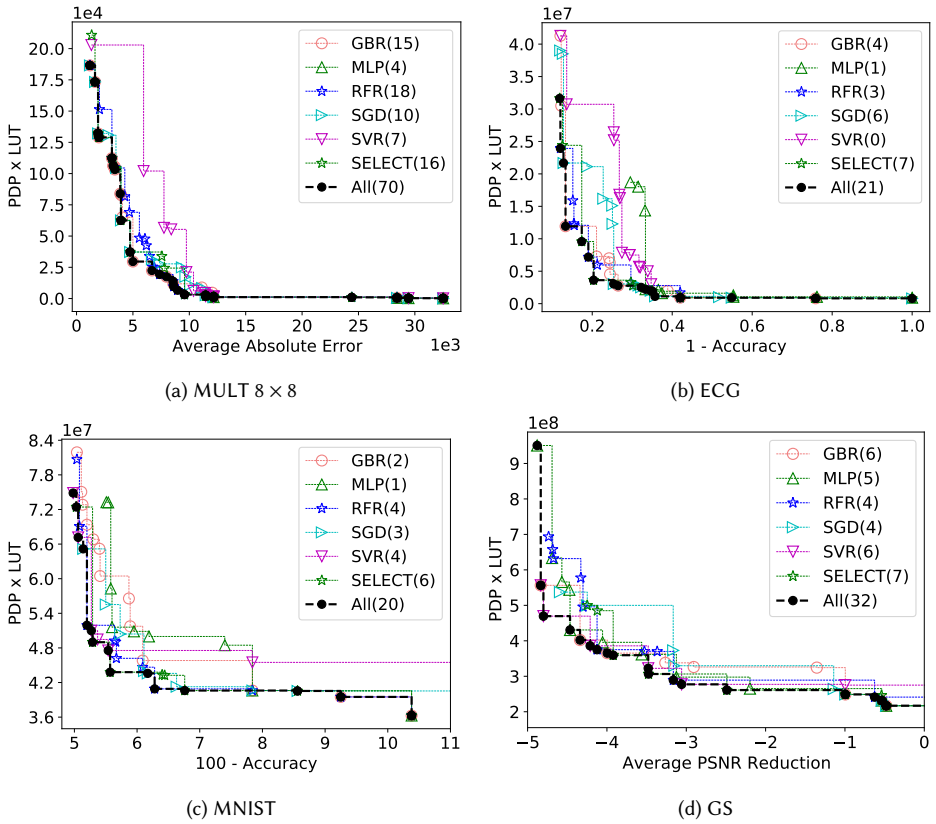


Fig. 18. Evaluated Pareto Front (EPF) for the design space exploration results for 8 × 8 multiplier and different applications using ML models. 'All' refers to the combined Pareto-front.

AppAxO_TRN, that were used for the training of the ML models. One of the corner-case designs is the configuration '1111111111111111111111111111111111111111', where all LUTs are enabled for the multiplier. The remaining 11 corner-case configurations were selected intuitively from the insight of the application-level analysis of 4 × 4 multipliers. For example, the application-level analysis of 4 × 4 multipliers had a few non-dominated design points where only a single LUT was enabled for the deployed multiplier. Therefore, for *AppAxO_TRN*, we had selected such 8 × 8 multiplier configurations where only one, two, or three LUTs were enabled (either at the most significant or least significant locations). For example, the 8 × 8 multiplier configurations '1000000000000000000000000000000000000000' and '0000000000000000000000000000000000000001' are two such configurations with one LUT enabled at the most significant and least significant locations, respectively.

Finally, *AppAxO_ML* denotes the set of design points generated using the proposed ML-based DSE. The design points for AppAxO_ML includes the collection of predicted Pareto-front points from the DSE runs corresponding to GBR, MLP, RFR, SGD, SVR and SELECT. It can be observed that in all cases, the ML-based DSE results in higher hypervolume and more number of Pareto-front points. The combined Pareto-fronts included 9, 4, 3, 2 design points with AppAxO_RND, 14, 6, 7 and 3 design points with AppAxO_TRN, and 25, 16, 13 and 19 design points with AppAxO_ML for MULT 8 × 8, ECG, MNIST and GS respectively.

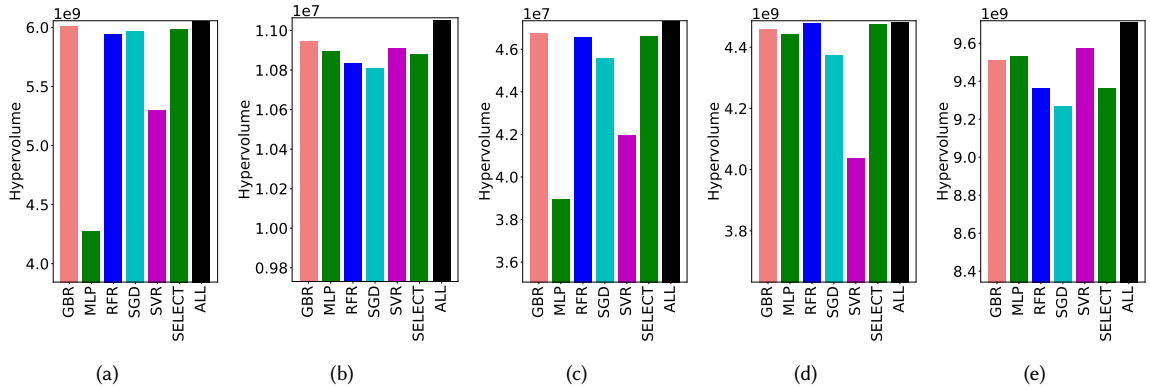


Fig. 19. Comparison of the hypervolume for the design space exploration results for 8×8 multiplier and different applications using ML models. 'ALL' refers to the hypervolume of the combined Pareto-front. (a) MULT 8×8 with Average Absolute Error as the accuracy metric (b) MULT 8×8 with Average Absolute Relative Error as the accuracy metric (c) ECG (d) MNIST (e) GS

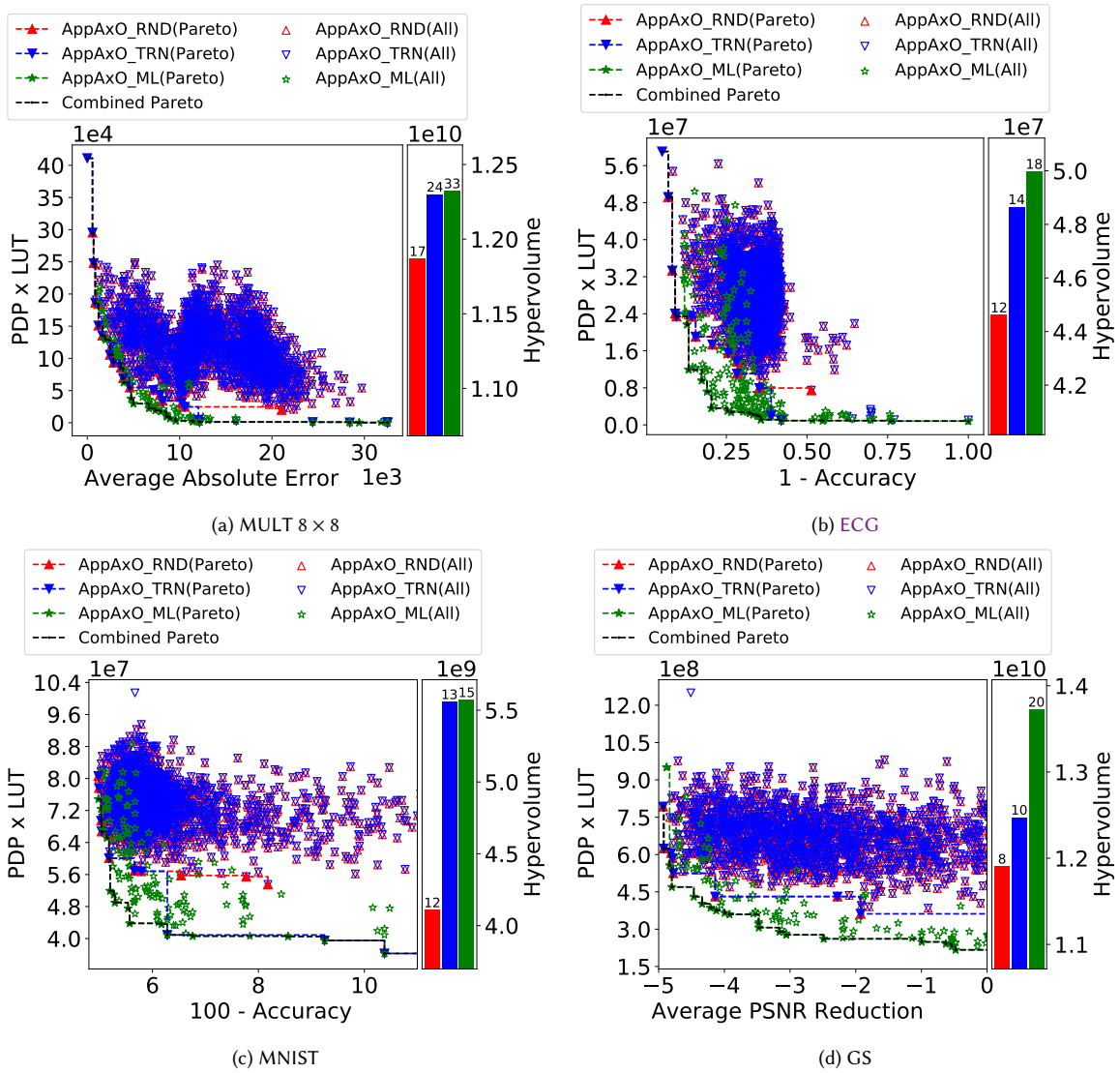
It can be observed that in some cases (MULT and MNIST), AppAxO_ML did not result in significant improvements over AppAxO_TRN. In both these cases, the intuitively added corner-case design points sufficed in providing the requisite accuracy-performance trade-offs. However, in other cases, AppAxO_ML succeeded in providing considerably better Pareto-front design points. Therefore, AppAxO_ML can aid the designer in searching for design points, beyond the more generic corner-case designs, that exploit the application's inherent error-tolerance. Moreover, unlike in the case of AppAxO_MBO (Figure 15), the candidate solutions generated by AppAxO_ML for true characterization are closer to the final Pareto-front.

5.7 Proposed Approximate Operators

5.7.1 Approximate Adders. To evaluate the efficacy of the *AppAxO*'s modeling of approach for generating approximate arithmetic operators, Figure 21 compares the hardware performance metrics and accuracy of *AppAxO*-generated 12-bit approximate unsigned adders with the 32 unsigned designs of the ApproxFPGAs library [27]. For a fair comparison, we have re-synthesized and re-implemented all the 12-bit approximate adders of ApproxFPGAs[9]. For a 12-bit accurate adder, *AppAxO* generates a total of 4095 corresponding approximate adders. For assessing the performance of the designs, we have used the $PDP \times LUT$ metric to incorporate all design metrics. The Pareto front analysis in Figure 21 shows that *AppAxO*-generated designs have more hypervolume contribution for all comparisons. For example, Figure 21(a) shows that *AppAxO*-generated non-dominated points have 1.11% more hypervolume contribution than the ApproxFPGAs non-dominated design points. Further, the numbers on the hypervolume bar show the individual non-dominated design points of each library. For example, the Pareto front analysis of only the *AppAxO* designs in Figure 21(a) shows a total of 47 non-dominated design points. Similarly, the Pareto front analysis of ApproxFPGAs provides 14 non-dominated design points. These results validate the efficacy of the *AppAxO* methodology in generating new approximate designs that can be selected according to an application's accuracy and performance requirements.

Further, the LUTs and carry chains specific approximation methodology of *AppAxO* enable better packing efficiency of the FPGA resources (CLB utilization) than ApproxFPGAs [27]. For example, to compare the packing efficiency of *AppAxO* generated 8-bit adders with the 8-bit designs of ApproxFPGAs, we implemented two different versions of a

[9]Available at https://github.com/ehw-fit/approx-fpgas



(a) MULT 8 × 8

(b) ECG

(c) MNIST

(d) GS

Fig. 20. Comparison of DSE results from AppAxO_ML against that from randomly generated points (AppAxO_RND) and design points used for training and testing of the ML models (AppAxO_TRN). Additional points synthesized, based on the Pareto-front points reported by AppAxO_ML, for each case: (a) 115 (b) 216 (c) 154 (d) 146

400-bit vector-adder utilizing 50 instances of 8-bit adders. In the first version, we utilized accurate adders, and in the second version, we utilized adders having comparable accuracy. Further, we have utilized a Virtex UltraScale FPGA for these experiments to accommodate a large number of inputs and outputs of the vector-adder. Compared to a Virtex-7 FPGA, the UltraScale FPGA provides an 8-bit wide carry-chain in a CLB[10]. The results of these experiments are shown in Table 6. As can be observed that for both experiments, *AppAxO* generated adders provide better packing efficiency

[10]The utilization of a different FPGA architecture also shows the efficacy of our proposed framework for different FPGA architectures.

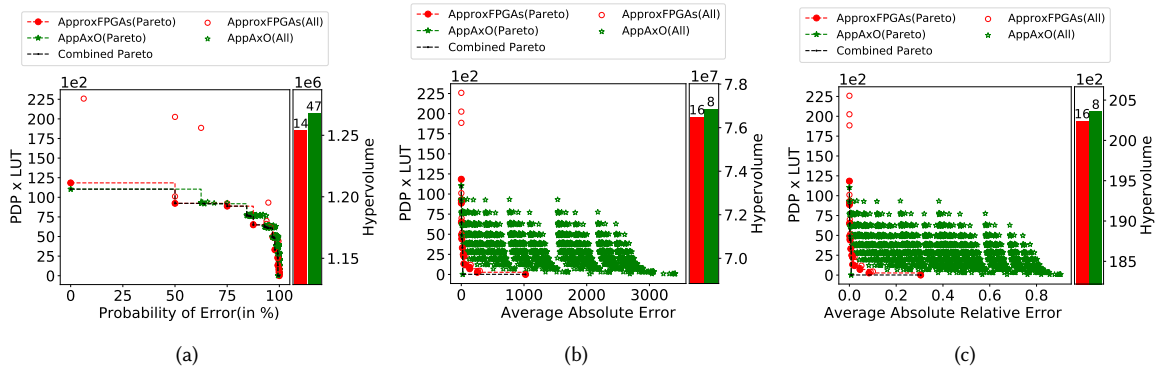


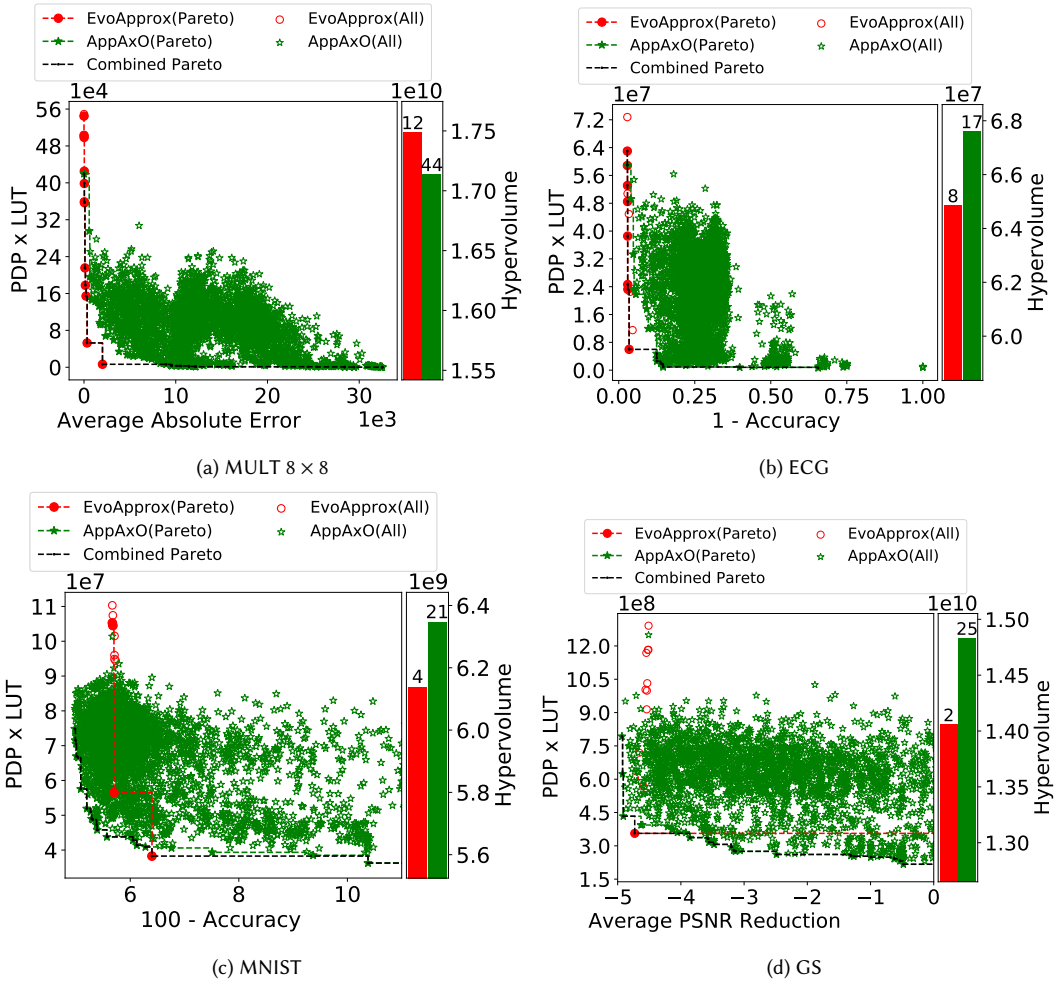
Fig. 21. Accuracy and performance comparison of *AppAxO*-generated unsigned adders with the approximate adders from ApproxFPGAs [27]

Table 6. Comparison of packing efficiency for implementing a vector-adder using 8-bit adders provided by *AppAxO* and ApproxFPGAs [27]

Design	Adder	Single Adder LUTs	Average Absolute Error	Vector-Adder LUTs	CLB Utilization
AppAxO	Adder_255	8	0.0	400	50
ApproxFPGAs	add8u_0FP	8	0.0	400	137
AppAxO	Adder_063	6	3.0	300	50
ApproxFPGAs	add8u_0B1	5	2.8	250	91

by utilizing a fewer number of CLBs. Similar efficacy of *AppAxO* generated designs is observed in comparison to other designs of ApproxFPGAs.

5.7.2 Proposed Approximate Multipliers. The resulting design points obtained from all explorations related to *AppAxO*, AppAxO_TRN, AppAxO_MBO and AppAxO_ML, were compared with the multipliers proposed in EvoApprox. A total of 3987 8×8 multiplier design configurations were characterized for stand-alone and application-specific performance estimation. Figure 22 shows the comparison of the Pareto front obtained for standalone multipliers and approximate multipliers used in the three test applications. It can be observed that the hypervolume of *AppAxO* is lower than that of EvoApprox in the stand-alone multipliers. However, in the search for application-specific multipliers, we report considerable improvements with *AppAxO* for all three applications. If we consider the combination of EvoApprox and *AppAxO*, we observed 10, 18, and 20 Pareto-front points with *AppAxO* compared to 8, 1, and 1 points with EvoApprox for ECG, MNIST, and GS, respectively. Figure 23 shows the comparison of the behavioral accuracy and the accelerator's performance, each metric considered separately, for AppAxO and EvoApprox multipliers implemented for ECG. The AppAxO-generated multipliers show better quality Pareto-front design points across all the three metrics. For LUT utilization, the combine Pareto-front included 14 design points with 4 and 10 points from EvoApprox and AppAxO respectively. Similarly, for CPD and power dissipation the combined Pareto front had 14 (EvoApprox: 3, AppAxO: 11) and 17 (EvoApprox: 8, AppAxO: 9) design points. It must be noted that although we show the results for each accelerator metric separately, the optimization problem used to generate these points was still based on the minimization of $PDP \times LUT$.



(a) MULT 8 × 8

(b) ECG

(c) MNIST

(d) GS

Fig. 22. Comparison of the proposed multipliers in EvoApprox [21] to that of *AppAxO*. The design points for *AppAxO* were obtained obtained across different exploration methods including randomized search, AppAxO_MBO and AppAxO_ML.

6 CONCLUSION

Most state-of-the-art approximate arithmetic operators follow an application-agnostic design methodology. These operators do not have a generic approximation methodology to implement new approximate designs for an application's changing accuracy and performance requirements. We address these limitations in this paper by presenting the *AppAxO* methodology. *AppAxO* presents a generic methodology for designing approximate operators according to an input configuration defined by an application. The configuration defines the number of active LUTs involved in partial product generation. For this purpose, *AppAxO* utilizes an MBO-based technique that produces only those multiplier configurations which can satisfy an application's accuracy and performance constraints. Our methodology also deploys various ML models to use GA to explore the large design space of individual multipliers and their utilization in various applications by estimating the behavioral accuracy and corresponding performance gains. Compared to state-of-the-art

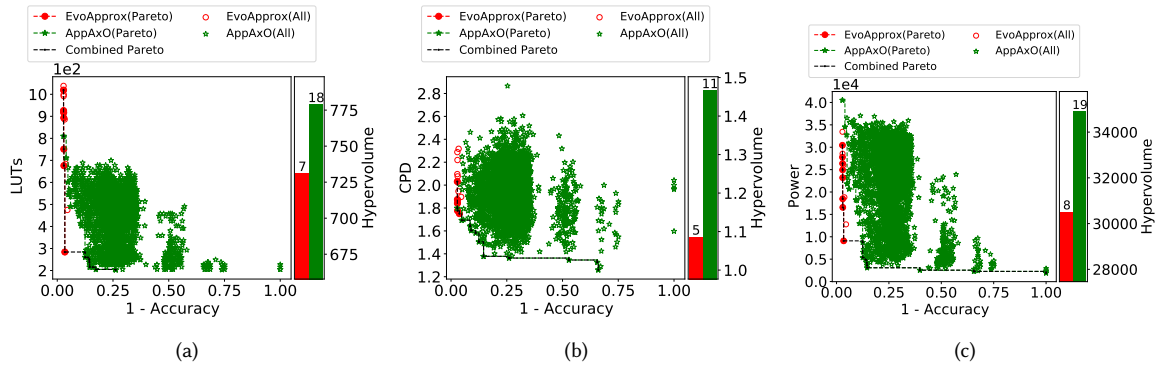


Fig. 23. Comparing accelerator performance and behavioral accuracy of using EvoApprox and *AppAxO*-generated 8 × 8 signed multipliers for ECG. (a) LUT utilization (b) Critical path delay (in ns) (c) Power dissipation (in μW)

approximate multipliers, *AppAxO* generates non-dominated design points with more hypervolume contribution for various applications. *AppAxO* methodology is generic and can be applied to any arithmetic circuit utilizing LUTs and the carry chains for its implementation.

By defining a LUT-like building block for ASIC-based systems, *AppAxO* can be extended to design approximate arithmetic operators for ASICs. To this end, one of the possible solutions is to utilize binary strings to address the various utilized gates in a standard cell-based design. The actual values of the different bit fields in the binary string will denote the activation/deactivation of particular gates. One of the potential challenges in this approach is handling the connections between various gates in the case of a deactivated intermediate gate. Another approach for utilizing AppAxO methodology for designing ASIC-based application-specific approximate operators can be utilizing standard cell gates to implement small computational units. A computational unit will represent the functionality of an FPGA-based LUT and carry-chain cell. These computational units can be utilized to implement various circuits. However, a single 6-input LUT in an FPGA can represent any 6-input combinational function. Therefore, in this approach, a designer may need to implement a large number of computational units to represent the circuit under test. Further, a single gate in a standard cell library can have different architectures for performance, area, and power tradeoffs. The availability of various gate architectures can further increase the overall complexity of designing ASIC-based application-specific approximate operators.

7 ACKNOWLEDGMENTS

This work is supported by the German Research Foundation (DFG) funded Project ReAp under Grant 380524764.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.

[2] Anonymous. 2021. MNIST-cnn. https://github.com/integeruser/MNIST-cnn

[3] Chidanand Apté and Sholom Weiss. 1997. Data mining with decision trees and decision rules. *Future generation computer systems* 13, 2-3 (1997), 197–210.

[4] Charles R Baugh and Bruce A Wooley. 1973. A two's complement parallel array multiplication algorithm. *IEEE Transactions on computers* 100, 12 (1973), 1045–1047.

[5] Francesco Biscani and Dario Izzo. 2020. A parallel global multiobjective framework for optimization: pagmo. *Journal of Open Source Software* 5, 53 (2020), 2338. https://doi.org/10.21105/joss.02338

[6] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. Springer, 177–186.

[7] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Analysis and Characterization of Inherent Application Resilience for Approximate Computing. In *Proceedings of the 50th Annual Design Automation Conference* (Austin, Texas) *(DAC '13)*. Association for Computing Machinery, New York, NY, USA, Article 113, 9 pages. https://doi.org/10.1145/2463209.2488873

[8] Gari D Clifford, Chengyu Liu, Benjamin Moody, Li-wei H. Lehman, Ikaro Silva, Qiao Li, A E Johnson, and Roger G. Mark. 2017. AF classification from a short single lead ECG recording: The PhysioNet/computing in cardiology challenge 2017. In *2017 Computing in Cardiology (CinC)*. 1–4. https://doi.org/10.22489/CinC.2017.065-469

[9] Li Deng. 2012. The MNIST database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.

[10] Zahra Ebrahimi, Salim Ullah, and Akash Kumar. 2020. SIMDive: Approximate SIMD Soft Multiplier-Divider for FPGAs with Tunable Accuracy. In *Proceedings of the 2020 on Great Lakes Symposium on VLSI* (Virtual Event, China) *(GLSVLSI '20)*. Association for Computing Machinery, New York, NY, USA, 151–156. https://doi.org/10.1145/3386263.3406907

[11] Jerome H Friedman. 2002. Stochastic gradient boosting. *Computational statistics & data analysis* 38, 4 (2002), 367–378.

[12] Vaibhav Gupta, Debabrata Mohapatra, Anand Raghunathan, and Kaushik Roy. 2013. Low-Power Digital Signal Processing Using Approximate Adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 1 (2013), 124–137. https://doi.org/10.1109/TCAD.2012.2217962

[13] Soheil Hashemi, R Iris Bahar, and Sherief Reda. 2015. DRUM: A dynamic range unbiased multiplier for approximate applications. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 418–425.

[14] Hou-Jen Ko and Shen-Fu Hsiao. 2011. Design and Application of Faithfully Rounded and Truncated Multipliers With Combined Deletion, Reduction, Truncation, and Rounding. *IEEE Transactions on Circuits and Systems II: Express Briefs* 58, 5 (2011), 304–308. https://doi.org/10.1109/TCSII.2011.2148970

[15] Parag Kulkarni, Puneet Gupta, and Milos Ercegovac. 2011. Trading Accuracy for Power with an Underdesigned Multiplier Architecture. In *2011 24th Internatioal Conference on VLSI Design*. 346–351. https://doi.org/10.1109/VLSID.2011.51

[16] Khaing Yin Kyaw, Wang Ling Goh, and Kiat Seng Yeo. 2010. Low-power high-speed multiplier for error-tolerant application. In *2010 IEEE international conference of electron devices and solid-state circuits (EDSSC)*. IEEE, 1–4.

[17] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791

[18] Andy Liaw, Matthew Wiener, et al. 2002. Classification and regression by randomForest. *R news* 2, 3 (2002), 18–22.

[19] Sparsh Mittal. 2016. A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.* 48, 4, Article 62 (March 2016), 33 pages. https://doi.org/10.1145/2893356

[20] Vojtech Mrazek, Muhammad Abdullah Hanif, Zdenek Vasicek, Lukas Sekanina, and Muhammad Shafique. 2019. AutoAx: An Automatic Design Space Exploration and Circuit Building Methodology Utilizing Libraries of Approximate Components. In *Proceedings of the 56th Annual Design Automation Conference 2019* (Las Vegas, NV, USA) *(DAC '19)*. Association for Computing Machinery, New York, NY, USA, Article 123, 6 pages. https://doi.org/10.1145/3316781.3317781

[21] Vojtech Mrazek, Radek Hrbacek, Zdenek Vasicek, and Lukas Sekanina. 2017. EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 258–261. https://doi.org/10.23919/DATE.2017.7926993

[22] Vojtech Mrazek, Syed Shakib Sarwar, Lukas Sekanina, Zdenek Vasicek, and Kaushik Roy. 2016. Design of Power-Efficient Approximate Multipliers for Approximate Artificial Neural Networks. In *Proceedings of the 35th International Conference on Computer-Aided Design* (Austin, Texas) *(ICCAD '16)*. Association for Computing Machinery, New York, NY, USA, Article 81, 7 pages. https://doi.org/10.1145/2966986.2967021

[23] Vojtech Mrazek, Lukas Sekanina, and Zdenek Vasicek. 2020. Libraries of Approximate Circuits: Automated Design and Application in CNN Accelerators. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 10, 4 (2020), 406–418. https://doi.org/10.1109/JETCAS.2020.3032495

[24] Fionn Murtagh. 1991. Multilayer perceptrons for classification and regression. *Neurocomputing* 2, 5-6 (1991), 183–197.

[25] Jiapu Pan and Willis J. Tompkins. 1985. A Real-Time QRS Detection Algorithm. *IEEE Transactions on Biomedical Engineering* BME-32, 3 (1985), 230–236. https://doi.org/10.1109/TBME.1985.325532

[26] Nicola Petra, Davide De Caro, Valeria Garofalo, Ettore Napoli, and Antonio G. M. Strollo. 2010. Truncated Binary Multipliers With Variable Correction and Minimum Mean Square Error. *IEEE Transactions on Circuits and Systems I: Regular Papers* 57, 6 (2010), 1312–1325. https://doi.org/10.1109/TCSI.2009.2033536

[27] Bharath Srinivas Prabakaran, Vojtech Mrazek, Zdenek Vasicek, Lukas Sekanina, and Muhammad Shafique. 2020. ApproxFPGAs: Embracing ASIC-Based Approximate Arithmetic Components for FPGA-Based Systems. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. https://doi.org/10.1109/DAC18072.2020.9218533

[28] Bharath Srinivas Prabakaran, Semeen Rehman, Muhammad Abdullah Hanif, Salim Ullah, Ghazal Mazaheri, Akash Kumar, and Muhammad Shafique. 2018. DeMAS: An efficient design methodology for building approximate adders for FPGA-based systems. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 917–920. https://doi.org/10.23919/DATE.2018.8342140

[29] Semeen Rehman, Walaa El-Harouni, Muhammad Shafique, Akash Kumar, Jorg Henkel, and Jörg Henkel. 2016. Architectural-space exploration of approximate multipliers. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/2966986.2967005

[30] Muhammad Shafique, Waqas Ahmad, Rehan Hafiz, and Jörg Henkel. 2015. A Low Latency Generic Accuracy Configurable Adder. In *Proceedings of the 52nd Annual Design Automation Conference* (San Francisco, California) *(DAC '15)*. Association for Computing Machinery, New York, NY, USA, Article 86, 6 pages. https://doi.org/10.1145/2744769.2744778

[31] Alex J Smola and Bernhard Schölkopf. 2004. A tutorial on support vector regression. *Statistics and computing* 14, 3 (2004), 199–222.

[32] Salim Ullah, Sanjeev Sripadraj Murthy, and Akash Kumar. 2018. SMApproxlib: Library of FPGA-Based Approximate Multipliers. In *Proceedings of the 55th Annual Design Automation Conference* (San Francisco, California) *(DAC '18)*. Association for Computing Machinery, New York, NY, USA, Article 157, 6 pages. https://doi.org/10.1145/3195970.3196115

[33] Salim Ullah, Tuan Duy Anh Nguyen, and Akash Kumar. 2021. Energy-Efficient Low-Latency Signed Multiplier for FPGA-Based Hardware Accelerators. *IEEE Embedded Systems Letters* 13, 2 (jun 2021), 41–44. https://doi.org/10.1109/les.2020.2995053

[34] Salim Ullah, Semeen Rehman, Bharath Srinivas Prabakaran, Florian Kriebel, Muhammad Abdullah Hanif, Muhammad Shafique, and Akash Kumar. 2018. Area-Optimized Low-Latency Approximate Multipliers for FPGA-Based Hardware Accelerators. In *Proceedings of the 55th Annual Design Automation Conference* (San Francisco, California) *(DAC '18)*. Association for Computing Machinery, New York, NY, USA, Article 159, 6 pages. https://doi.org/10.1145/3195970.3195996

[35] Salim Ullah, Semeen Rehman, Muhammad Shafique, and Akash Kumar. 2021. High-Performance Accurate and Approximate Multipliers for FPGA-based Hardware Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021), 1–1. https://doi.org/10.1109/TCAD.2021.3056337

[36] Salim Ullah, Hendrik Schmidl, Siva Satyendra Sahoo, Semeen Rehman, and Akash Kumar. 2021. Area-Optimized Accurate and Approximate Softcore Signed Multiplier Architectures. *IEEE Trans. Comput.* 70, 3 (2021), 384–392. https://doi.org/10.1109/TC.2020.2988404

[37] Xilinx. 2017. UltraScale Architecture Configurable Logic Block.

[38] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. 2017. AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE Design Test* 34, 2 (2017), 60–68. https://doi.org/10.1109/MDAT.2016.2630270