Adaptive Approximate Accelerators with Controlled Quality Using Machine Learning

Mahmoud Masadeh, Osman Hasan, and Sofiène Tahar

1 Introduction

The ongoing scaling in feature size has caused integrated circuit (IC) behavior vulnerable to soft errors as well as process, voltage, and temperature variations. Thus, the challenge of assuring strictly exact computing is increasing [1]. On the other hand, present-age computing systems are pervasive, portable, embedded, and mobile, which led to an ever-increasing demand for ultra-low power consumption, small footprint, and high-performance systems. Such battery-powered systems are the main pillars of the internet of things (IoT), which do not necessarily need entirely accurate results.

Approximate computing (AC), known as best-effort computing, is a nascent computing paradigm that allows us to achieve these objectives by compromising the arithmetic accuracy [2]. Nowadays, many applications, such as image processing, multimedia, recognition, machine learning, communication, big data analysis, and data mining, are error-tolerant and thus can benefit from approximate computing. These applications exhibit *intrinsic error resilience* due to the following factors [3]: (i) redundant and noisy input data, (ii) lack of golden or single output, (iii) imperfect

M. Masadeh (✉)
Computer Engineering Department, Yarmouk University, Irbid, Jordan
e-mail: mahmoud.s@yu.edu.jo

O. Hasan
Electrical Engineering Department, National University of Sciences and Technology, Islamabad, Pakistan
e-mail: osman.hasan@seecs.nust.edu.pk

S. Tahar
Department of Electrical and Computer Engineering, Concordia University Montreal, Montreal, QC, Canada
e-mail: tahar@ece.concordia.ca

W. Liu et al. (eds.), *Design and Applications of Emerging Computer Systems*,
https://doi.org/10.1007/978-3-031-42478-6_19

perception in the human sense, and (iv) the usage of implementation algorithms with self-healing and error attenuation patterns.

Different approximation strategies, which fall under the umbrella of approximate computing, e.g., the voltage over scaling [4], algorithmic approximations [5], and approximation of basic arithmetic operations [6], have gained a significant research interest, in both academia and industry, such as IBM [7], Intel [8], and Microsoft [9]. However, approximate computing is still immature and does not have standards yet, which poses severe bottlenecks and main challenges. Thus, future work of AC should be guided by the following general principles to achieve the best efficiency [3]:

1. Significance-driven approximation: Identifying the approximable parts of an application or circuit design is a great challenge. Therefore, it is critical to distinguish the approximable parts with their approximation settings.
2. Measurable notion of approximation quality: Quality specification and verification of approximate design are still open challenges, where quality metrics are application and user-dependent. To quantify approximation errors, various quality metrics are used.
3. Quality configuration: Error resiliency of applications depends on the applied inputs and the context in which the outputs are consumed.
4. Asymmetric approximation benefits: It is essential to identify the approximable components of the design, which reduces the quality insignificantly while improving efficiency considerably.

For a static approximate design, the approximation error continues during its operational lifetime. It restricts approximation versatility and results in under- or over-approximated systems for dynamic input data, causing excessive power usage and insufficient accuracy, respectively. Given the dynamic nature of the applied inputs into static approximate designs, errors are the norm rather than the exception in approximate computing, where the error magnitude depends on the user inputs [10]. On the other hand, the defined tolerable error threshold, i.e., target output quality (TOQ), can be dynamically changed. In both cases, errors with a high value produced by approximate components in an approximate accelerator, even with a low error rate, have a more significant impact on the quality than those caused by approximate parts with a small magnitude. This is in line with the notion of fail-small, fail-rare, or fail-moderate approaches, [11], where error magnitudes and rates should be restricted to avoid high loss in the output quality. The fail-small technique allows approximations with low error magnitudes, while the fail-rare technique allows approximations with low error rates. On the other hand, the fail-moderate technique allows approximations with moderate error magnitude and moderate error rate [12]. Thus, the approaches mentioned above limit the design space to prevent approximations with high error rates and high error magnitudes, where such a combination degrades the quality loss significantly.

Quality assurance of approximate computing is still missing a mathematical model for the impact of approximation on the output quality [3]. Toward this goal, in this chapter, we develop a runtime adaptive approximate accelerator. For that, we utilize a set of energy-efficient approximate multipliers which we designed in [13]. The adaptive design is based on fine-grained input data to satisfy a user-defined target output quality (TOQ) constraint. Design adaptation uses a machine learning-based design selector to dynamically choose the most suitable approximate design for runtime data. The target approximate accelerator is implemented with configurable levels and types of approximate multipliers.

1.1 Approximate Computing Error Metrics

Approximation introduces accuracy as a new design metric. Thus, several application-dependent error metrics are used to quantify approximation errors and evaluate design accuracy [14]. For example, considering an approximate design with two inputs, i.e., X and Y, of n-bit each, where the exact result is (P) and the approximate result is (P'), these error metrics include:

- Error Distance (ED): The arithmetic difference between the exact output and the approximate output for a given input, which is presented by $ED = |P - P'|$.
- Error Rate (ER): Also called error probability, which is the percentage of erroneous outputs among all outputs.
- Mean Error Distance (MED): The average of ED values for a set of outputs obtained by applying a set of inputs. MED is a useful metric for measuring the implementation accuracy of multiple-bit circuit design.
- Normalized Error Distance (NED): The normalization of MED by the maximum result that an exact design can have (P_{Max}). NED is an invariant metric independent of the size of the circuit. Therefore, it is used for comparing circuits of different sizes, and it is expressed as:
- Relative Error Distance (RED): The ratio of ED to the accurate output, which equals $RED = ED/P$.
- Mean Relative Error Distance (MRED): The average value of all possible relative error distances (RED).
- Mean Square Error (MSE): It is defined as the average of the squared ED values.
- Peak Signal-to-Noise Ratio (PSNR): The peak signal-to-noise ratio is a fidelity metric used to measure the quality of the output images; it indicates the ratio of the maximum pixel intensity to the distortion.

The presented metrics are not mutually exclusive, where one application may use several quality metrics.

1.2 Approximate Accelerators

Hardware accelerators are special hardware, which is devoted for executing frequently called functions. Accelerators are more efficient than software running on general-purpose processors. Generally, they are constructed by connecting multiple simple arithmetic modules. The existing literature has proposed the design of approximate accelerators using neural networks [15] or approximate functional units, particularly approximate adders [16] and multipliers [17]. Moreover, several functionally approximate designs for basic arithmetic modules, including adders [6], dividers [18], and multipliers [19], have been investigated for their pivotal role in various applications. These individually designed components are rarely used alone, especially in computationally intensive error-tolerant applications, which are amenable to approximation. The optimization of accuracy performance at the accelerator level has received little or no attention in the previous literature. Generally, hardware accelerators are constructed by connecting multiple simple arithmetic modules. For example, discrete Fourier transform (DFT) and discrete cosine transform (DCT) modules are used in signal and image processing. Approximate multipliers and multiply-accumulate units (MACs) are intensively used to build approximate accelerators.

Multipliers are one of the most foundational components for most functions and algorithms in classical computing. However, they are the most energy-costly units compared to other essential CPU functions such as register shifts or binary logical operators. Thus, their approximation would introduce an enhancement in their performance and energy, which automatically induces crucial benefits for the whole application. Approximate multipliers have been mainly designed using three techniques:

(i) Approximation in partial product generation: For example, Kulkarni et al. [20] proposed an approximate 2×2 binary multiplier at the gate level by changing a single entry in the Karnaugh map with an error rate of $1/16$.
(ii) Approximation in partial product tree: For example, error-tolerant multipliers (ETM) [21] divide the input operands into two parts, i.e., the multiplication part for the MSBs and the non-multiplication part for the LSBs, thus omitting the generation of some partial products [19].
(iii) Approximation in partial product summation: Approximate full adder (FA) cells are used to form an array multiplier, e.g., in [22], the approximate mirror adder has been used to develop a multiplier.

We focus on array multipliers, which are not the fastest neither the smallest. Their short wiring gives them a periodic structure with a compact hardware layout. Thus, they are one of the most used in embedded system on chip (SoC). In [23] and [24], we designed various 8- and 16-bit approximate array multipliers based on approximation in partial product summation.

1.3 Quality Control of Approximate Accelerators

Managing the quality of approximate hardware designs for dynamically changing inputs has substantial significance to guarantee that the obtained results satisfy the required target output quality (TOQ). To the best of our knowledge, there are very few works targeting the assurance of the accuracy of approximate systems compared to designing approximate components. While most prior works focus on *error prediction*, we propose to overcome the approximation error through an input-dependent *self-adaptation of design*.

Mainly, there are two approaches for monitoring and controlling the accuracy of the results of approximate accelerators at runtime. The first approach suggests to periodically, through *sampling techniques*, measure the error of an accelerator through comparing its outcome with the exact computation performed by the host processor. Then, a re-calibration and adjustment process is performed to improve the quality in subsequent invocations of the accelerator if the error is found to be above a defined range, e.g., Green [25] and SAGE [26]. However, the quality of unchecked invocations cannot be ensured, and the previous quality violations cannot be compensated. The second approach relies on implementing lightweight pre-trained error predictors to expect if the invocation of an approximate accelerator would produce an unacceptable error for a particular input dataset [27, 28].

However, the works [25–27], and [28] mainly target controlling software approximation, i.e., loops and functions approximation, through program re-execution and thus are not applicable for hardware designs. Moreover, they ignore input dependencies and do not consider choosing an adequate design from a set of design choices. Overall, none of these state-of-the-art techniques exploits the potential of different settings of approximate computing and their adaptations based on a user-specified quality constraint to ensure the accuracy of the individual outputs, which is the main idea we propose. Design adaptation could be implemented in software-based systems by having different versions of the approximate code, while hardware-based systems rely on having various implementations for the functional units. However, concurrently having such functional units diminishes approximation benefits. Thus, dynamic partial reconfiguration (DPR) could be used to have only a single implementation of the design at any instance of time.

2 Proposed Methodology

We aim to assure the quality of approximation by design adaptation by predicting the most suitable settings of the approximate design to execute the inputs. The proposed method predicts the *design settings* based on the applied input data and user preferences, without losing the gains of approximations. We mostly consider the case of approximate accelerators built with approximate functional units such as approximate multipliers.

We propose a comprehensive methodology that handles the limitations of the current state of the art in terms of fine-grained input dependency, suitability for various approximate modules (e.g., adders, dividers, and multipliers), and applicability to both hardware and software implementations. Figure 1 provides a general overview of the proposed methodology for design adaptation. As shown in

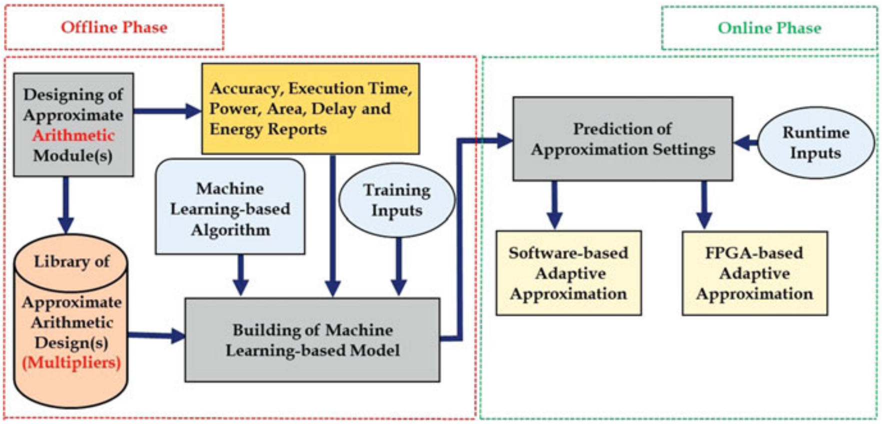
Offline Phase
Online Phase
Designing of Approximate Arithmetic Module(s)
Accuracy, Execution Time, Power, Area, Delay and Energy Reports
Machine Learning-based Algorithm
Training Inputs
Library of Approximate Arithmetic Design(s) (Multipliers)
Building of Machine Learning-based Model
Prediction of Approximation Settings
Runtime Inputs
Software-based Adaptive Approximation
FPGA-based Adaptive Approximation

Fig. 1 General overview of the proposed methodology

the figure, the methodology includes two phases: (1) The first is an offline phase, which is executed once for building a machine learning-based model. Such a model predicts the design settings. (2) The second is an online stage, where the machine learning-based model constantly accepts inputs and predicts accordingly based on the runtime inputs. The proposed methodology encompasses the following main steps:

(1) *Building a library of approximate designs*: The first step is designing the library of basic functional units, such as adders, multipliers, and dividers, with different settings, which will be integrated into a quality-assured approximate design. The characteristics of each design, e.g., accuracy, area, power, delay, and energy consumption, should be evaluated to highlight the benefits of approximation.
(2) *Building a machine learning-based model*: In the offline phase, we use supervised learning and employ decision trees (DT) and neural network (NN) algorithms to build a model to predict the unseen data, e.g., the design settings. This step incorporates generating and pre-processing the training data, such as quantization, sampling, and reduction. The training inputs are applied exhaustively to an approximate design to create the training data. For n-bit designs with two inputs, the size of the input combinations is 2^{2n}.
(3) *Predicting the approximation settings*: In the online phase, the user-specified runtime inputs, i.e., the target output quality (TOQ) and the inputs of the approximate design, are given to the ML-based models to predict approximation-related output, i.e., setting of the adaptive design. The implemented ML-based model should be lightweight, i.e., have a high prediction accuracy with fast execution.
(4) *Integrating the approximate accelerator into error-resilient applications*: For adaptive design, the approximate accelerator, which has been nominated by the ML-based model, is adapted within an error-resilient application. Such design

could be implemented in software (off-field-programmable gate array (FPGA), as explained in Sect. 3) or in hardware (on-FPGA, as described in Sect. 4).

Approximation approaches demand a *quality assurance* to adjust approximation settings/knobs and monitor the quality of fine-grained individual outputs. There are two approaches to adjusting the settings of an approximate program to ensure the quality of results:

(i) *Forward design* [29], which sets the design knobs and then observes the quality of results. However, the output quality of some inputs may reach unacceptable levels.
(ii) *Backward design* [30], which tries to locate the optimal knob setting for a given bound of output quality; this requires examining a large space of knob settings for a given input, which is unmanageable.

We present an *adaptive approximate design* that allows altering the settings of approximation, at runtime to meet the preferred output quality. The principal idea is to generate a machine learning-based input-aware *design selector*, which can adjust the approximate design based on the applied inputs, to meet the required quality constraints. Our technique is general in terms of quality metrics and supported approximate designs. It is primarily based on a library of 8- and 16-bit approximate multipliers with 20 different configurations and well-known power dissipation, performance, and accuracy profiles [13]. Moreover, we utilize a backward design approach to dynamically adjust the design to satisfy the desired target output quality (TOQ) based on machine learning (ML) models. The TOQ is a user-defined quality constraint, which represents the maximum permissible error for a given application. The proposed design flow is adaptable, i.e., applicable to approximate functional units other than multipliers, e.g., approximate multiply-accumulate units [31] and approximate meta-functions [32].

2.1 Machine Learning-Based Models

ML-based algorithms find solutions by learning through training data [33]. Supervised learning allows for a rapid, flexible, and scalable way to develop accurate models that are specific to the set of application inputs and TOQ. The error for an approximate design with particular settings can be predicted based on the applied inputs. In [34], we designed and evaluated various ML-based models, based on the analyzed data and several algorithms, developed in the statistical computing language R. These models express the design selector for the adaptive design. *Linear regression* (LR) models were found to be the simplest to develop; however, their accuracy is the lowest, i.e., around 7%. Thus, they are not suitable for our proposed methodology. On the other hand, *decision tree* (DT) models based on both *C5.0* and *rpart* algorithms achieve an accuracy of up to 64%, while *random forest* (RF) models, with an overhead of 25 decision trees, achieve an accuracy of up to

68%. The most accurate models are based on *neural networks*, but they suffer from long development time, design complexity, and high-energy overhead [27]. In this work, we implement and evaluate two versions of the *design selector*, based on decision tree and neural network models. Accordingly, we identify and select the most suitable one to implement in our methodology.

2.1.1 Decision Tree-Based Design Selector

The DT algorithm uses a flowchart-like tree layout to partition data into various predefined classes, thereby providing the description, categorization, and generalization of the given datasets [35]. Unlike the linear model, it models non-linear relationships quite well. Thus, it is used in a wide range of applications, such as credit risk of loans and medical diagnosis [36]. Decision trees are usually employed for *classification* over datasets, through recursively partitioning the data, such that observations with the same label are grouped [36].

Generally speaking, a decision tree model could be replaced by a lookup table (LUT) which contains all the training data that are used to build the DT model [34]. When searching the LUTs, we could use the first matched value, i.e., design settings that satisfy the TOQ, which could be a better solution obtained with a little search effort. For DT-based models, we do not need to specify which value to retrieve. However, it is possible to obtain a result which is closer to the TOQ by changing the settings of the tree such as (1) the maximum depth of any node of the tree, (2) the minimum number of observations that must exist in a node in order for a split to be attempted, and (3) the minimum number of observations in any terminal node. In general, for embedded and limited resource systems, a lookup table is not a viable solution if the number of entries becomes very large [37]. In fact, for a circuit with two 16-bit inputs, we need to generate 2^{32} input patterns to cover all possible scenarios of a circuit.

2.1.2 Neural Network-Based Design Selector

We implemented a two-step NN-based design selector by predicting the design *Degree* first (how much to approximate) and then the *Type* (which approximate full adder to use). The model for *Degree* prediction has an accuracy of 82.17%, while the four models for *Type* prediction have an average accuracy of 67.3%. These models have a single hidden layer with a *sigmoid* activation function.

3 Software-Based Adaptive Design of Approximate Accelerators

We present a detailed description of the proposed methodology for designing adaptive approximate accelerators, where the proposed design can be implementable in both software and hardware. This section shows the aspects of software-based implementation. Section 4 is devoted to the FPGA-based hardware implementation.

3.1 Adaptive Design Methodology

As shown in Fig. 2, the proposed methodology contains two phases: (1) an offline stage, where we build an ML-based model, and (2) an online stage, where we use the ML-based model based on the inputs to anticipate the settings of the adaptive design. The detailed steps of the presented methodology are:

(1) *Generating of Training Data*: Inputs are applied exhaustively to the approximate library to create the training data for building the ML-based model (design selector). For 8- and 16-bit designs, the size of the input combinations is 2^{16} and 2^{32}, respectively. Thus, a sampling of the training data could be used because it is impossible to generate an exhaustive training dataset for large circuits.

(2) *Clustering/Quantizing of Training Data*: Evaluating the design accuracy for a single input can provide the error distance (ED) metric only. However, mean error metrics (e.g., mean square error (MSE), peak signal-to-noise ratio (PSNR), and normalized error distance (NED)) are evaluated over a set of successively applied data rather than a scalar input. Thus, inputs with a specific distance from each other are considered a single cluster with the same estimated error metric. We propose to cluster every 16 consecutive input values. Based on that, each input for an 8-bit multiplier encompasses 16 clusters rather than 256 inputs. Similarly, for the 16-bit multiplier design, the number of clustered inputs is reduced to 2^{24} rather than 2^{32}.

(3) *Pre-processing of Training Data*: Inputs could be applied exhaustively for small circuits, e.g., 8-bit multipliers. However, the size of the input combinations for 16- and 32-bit designs is significant. Therefore, we have to reduce the size of the training data through sampling approaches to design a smaller and more efficient ML-based model. Moreover, for 16-bit designs, we prioritize the training data based on their area, power, and delay as well as accuracy and then reduce the training data accordingly.

(4) *Building of Machine Learning-Based Model*: We built decision trees and neural network-based models, which act as design selectors, to predict the most suitable settings of the design based on the applied inputs.

(5) *Selection of Approximate Design*: In the online phase, the user inputs, i.e., TOQ and inputs of the multiplier, are given to the ML-based models to predict the setting of the approximate design, i.e., *Type* of approximate components and *Degree* of approximation, which is utilized within an error-resilient application, e.g., image processing, in a software-based adaptive approximate execution.

The flow of the proposed methodology is depicted in Fig. 2. The main steps are done once offline. During the online phase, the user specifies the TOQ, where we build our models based on normalized error distance (NED) and peak signal-to-

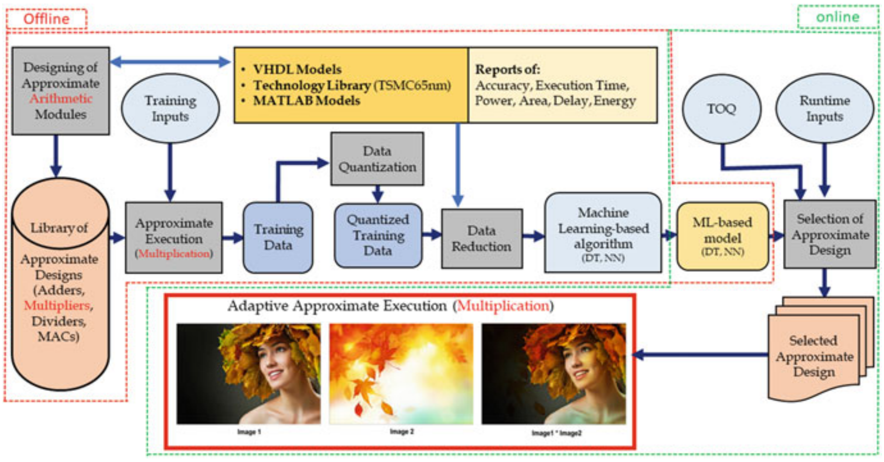
Offline
online
Designing of Approximate Arithmetic Modules
Training Inputs
• VHDL Models
• Technology Library (TSMC65nm)
• MATLAB Models
Reports of: Accuracy, Execution Time, Power, Area, Delay, Energy
TOQ
Runtime Inputs
Data Quantization
Library of Approximate Designs (Adders, Multipliers, Dividers, MACs)
Approximate Execution (Multiplication)
Training Data
Quantized Training Data
Data Reduction
Machine Learning-based algorithm (DT, NN)
ML-based model (DT, NN)
Selection of Approximate Design
Adaptive Approximate Execution (Multiplication)
Image 1
Image 2
Image1 * Image2
Selected Approximate Design

Fig. 2 A detailed methodology of software-based adaptive approximate design

noise ratio (PSNR) error metrics. An important design decision is to determine the configuration granularity, i.e., how much data to process before re-adapting the design, which is termed the window size (N). For example, in image processing applications, we select N to be equal to the size of colored components of an image. Then based on the length of inputs, i.e., L and N, we determine the number of times to reconfigure the design such that the final approximation benefits are still significant. After N inputs, a design adaptation is done, if any of the inputs or TOQ changes. The first step in such adaptation is input quantization, i.e., specifying the corresponding cluster for each input based on its magnitude, since design adaptation for every scalar input is impractical. To evaluate the inputs of an approximate design, various metrics, such as median, skewness, and kurtosis, have been used [38]. Thus, the input magnitude is the most suitable characteristic of design selection.

3.2 *Machine Learning-Based Models*

We developed a forward design-based model, as shown in Fig. 3a. The obtained accuracy for this model is 97.6% and 94.5% for PSNR and NED error metrics, respectively. Such high efficiency is due to the straightforward nature of the problem. However, we target the inverse design of finding the most suitable design settings (degree and type) for given inputs (C1 and C2) and error threshold, as shown in Fig. 3b.

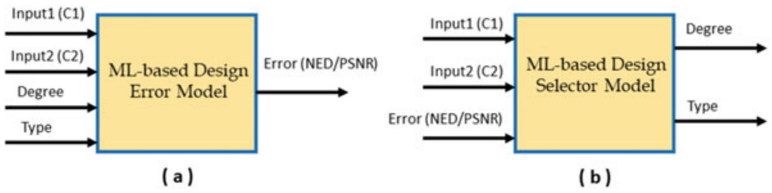

Fig. 3 Models for AC quality manager, (**a**) forward design and (**b**) inverse design

Table 1 Accuracy and execution time of DT- and NN-based design selectors

Model		Accuracy		Execution time (ms)	
Inputs	Output	DT	NN	DT	NN
C1, C2, PSNR	Degree	77.8%	82.17%	8.87	18.9
C1, C2, PSNR, s2=D1	Type	75.5%	66.52%	25.03	18.0
C1, C2, PSNR, s2=D2	Type	76.1%	70.21%	19.3	9.0
C1, C2, PSNR, s2=D3	Type	71.3%	73.22%	11.94	18.7
C1, C2, PSNR, s2=D4	Type	74.1%	59.08%	6.61	7.4

3.2.1 Decision Tree-Based Design Selector

Based on the error analysis of the approximate designs [39], we noticed that the error magnitude is correlated to the approximation *Degree* in a more significant manner than the design *Type*. Such correlation is evident in the accuracy of the models, where these models have an average accuracy of 77.8% and 74.3% for predicting the design *Degree* and *Type*, respectively, as shown in Table 1. The time for executing the *software* implementation of these models is very short, i.e., 24.6 ms in total with 8.87 ms to predict the design *Degree* and 15.72 ms to predict the design *Type*. This time is negligible compared to the time of running an application, such as image blending.

3.2.2 Neural Network-Based Design Selector

As shown in Table 1, the model for *Degree* prediction has an accuracy of 82.17%, while the four models for *Type* prediction have an average accuracy of 67.3%. The time for executing the *software* implementation of these models is short, i.e., 32.18 ms in total with 18.9 ms to predict the design *Degree* and 13.28 ms to predict the design *Type*. This time is negligible compared to the running time of an application, such as image processing. Compared to the DT-based model, the NN-based model has an execution time, which is 1.31× higher than the DT, while its average accuracy is almost 0.98× of the accuracy achieved by the DT-based model. Next, we evaluate the *software* implementation of the proposed methodology, which utilizes the DT-based design selector. We discard the NN-based design selector due to the absence of advantages over DT.

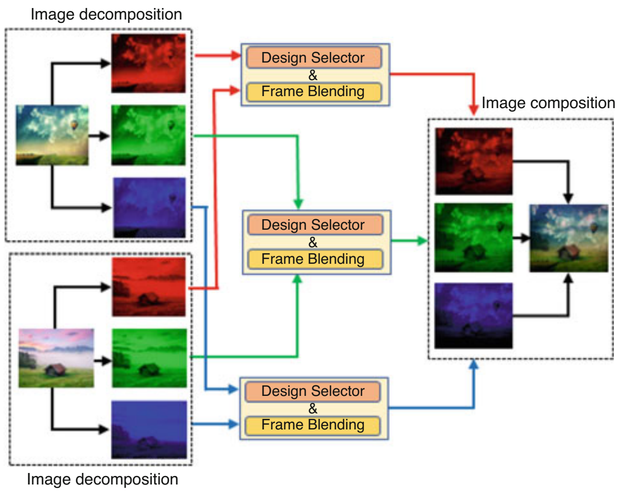

Fig. 4 Adaptive image/video blending at component level

3.3 Experimental Results of Image Blending

Here, we evaluate the effectiveness of the *software* implementation of the fully automated proposed system. We run MATLAB on a machine with 8 GB DRAM and an i5 CPU with a speed of 1.8 GHz. We assess the proposed methodology based on an image blending application, where we use a set of images. The *execution time* is a quality metric, where its overhead is relatively small compared to the original applications, as shown in the sequel.

Image blending in multiplication mode multiplies numerous images to look like a single image. For example, blending two-colored videos, each with N_f frames of size N_r rows by N_c columns per image, involves a total of $3 \times N_f \times N_r \times N_c$ pixels. Each image has three colored components/channels, i.e., red, green, and blue, where the values of their pixels are expected to differ. A static configuration uses a single design of an 8-bit multiplier to perform all multiplications, even when their pixels are different. Therefore, for improved output quality, we propose to adapt the approximate design per channel as shown in Fig. 4. However, for a video with a set of successive frames, e.g., 30 frames per second, the proposed methodology can be run for the first frame only since the other frames have very close pixel values. This way, the design selector continuously monitors the inputs and efficiently finds the most suitable design for each colored component to meet the required TOQ.

Various metrics, e.g., median, skewness, and kurtosis, have been used in the literature to represent the inputs of approximate designs [38]. However, their proposed approximate circuits heavily depend on the training data used during the approximation process. Since the error magnitude depends on the user inputs, we rely on pixel values to select a suitable design. However, setting the configuration

Table 2 Characteristics of the blended images

			Input 1 (Image1)			Input2 (Image2)		
Example	(Image1, Image2)	Frame characteristic	Red	Green	Blue	Red	Green	Blue
1	(Frame, City)	Average	131	163	175	172	153	130
		Cluster	9	11	11	11	10	9
2	(Sky, Landscape)	Average	121	149	117	160	156	147
		Cluster	8	10	8	11	10	10
3	(Text, Whale)	Average	241	241	241	48	156	212
		Cluster	16	16	16	4	10	14
4	(Girl, Beach)	Average	177	158	140	168	176	172
		Cluster	12	10	9	11	12	11
5	(Girl, Tree)	Average	102	73	40	239	193	118
		Cluster	7	5	3	16	13	8

granularity at the pixel level is impractical. On the other hand, the design selection per colored component is more suitable.

We compute the average of the pixels of each colored component to determine the most suitable design. Two completely different images may have the same average of their pixels. Unfortunately, this could result in the same selected approximate design. To avoid this scenario, we reduce the configuration granularity by dividing the colored component into multiple segments, e.g., four segments. Thus, we use various designs, rather than a single design, for each colored component. Next, we analyze the results of applying the proposed methodology on a set of ten images. The photos of each set are then blended at the component level, as shown in Fig. 4, to evaluate the efficiency of the proposed methodology.

We use a set of ten different images, each of size $N_r \times N_c = 250 \times 400 = 10^5$ pixels, and each image is segmented into three colored components. Table 2 shows the average values of the pixels of each colored component and the associated input cluster, which are denoted as *Average* and *Cluster*, respectively.

We target 49 different values of TOQ, i.e., PSNR ranges from 17 dB to 65 dB, for each blending example. Thus, we run the methodology 245 times, i.e., 5×49. For every invocation, based on the corresponding cluster for each input, i.e., $C1$ and $C2$, and the associated target PSNR, 1 of the 20 used designs is selected and used for blending. For illustration purposes, in the sequel, we explain *Example5* in detail. As shown in Table 2, the *Girl* image has a red component with an average of 102, which belongs to *Cluster 7*, i.e., $C1_R = 7$. Similarly, the *Tree* image has a red component with an average of 239, which belongs to *Cluster 16*, i.e., $C2_R = 16$. The green components belong to *Clusters* 5 and 13 ($C1_G = 5$, $C2_G = 13$), while the blue components belong to *Clusters* 3 and 8 ($C1_B = 3$, $C2_B = 8$). Then, we adapt the design by calling the design selector thrice, i.e., once for every colored component, assuming TOQ$=$ 17 dB. The selected designs are used, and the obtained quality is 16.9 dB, which is insignificantly less than the TOQ.

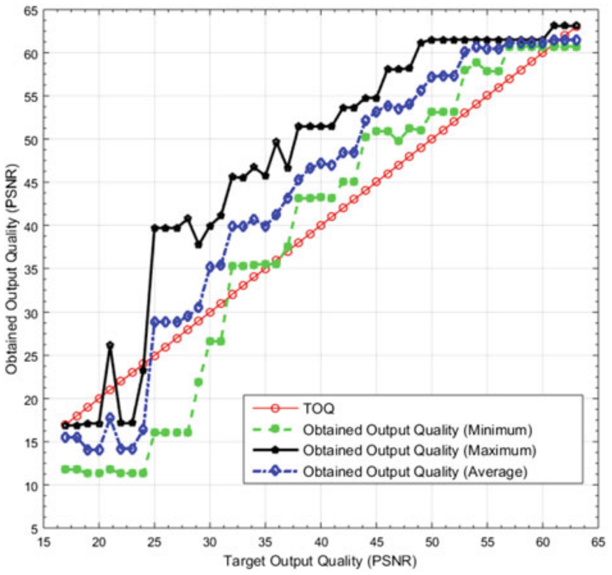

Fig. 5 Obtained output quality for image blending of *Set-1*

Accuracy Analysis of Adaptive Design Figure 5 shows the minimum, maximum, and average curves of the obtained output quality, each evaluated over five examples of image blending. Out of the 245 selected designs, 49 predicted designs are violating the TOQ, even insignificantly, i.e., the obtained output quality is below the red line. The unsatisfied output quality is attributed mainly to model imperfection. The best achievable prediction accuracy is based on the accuracy of the two models executed consecutively, i.e., *Degree* model with 77.8% and *Type* model with 76.1%. The accuracy of our model prediction is 80%, which is in agreement with the average accuracy of the DT-based models, as shown in Table 1.

Execution Time Analysis of Adaptive Design Figure 6 displays the average execution time of the 5 examples of image blending evaluated over 20 static designs. The shown time is normalized for the execution time of the exact design. All designs have a time reduction ranging from 1.8% to 13.6% with an average of 3.96%. For the five examples of image blending, we assessed the execution time of the adaptive design, where the target PSNR ranges between 17 dB and 65 dB for each case. Figure 7 shows the execution time for the 5 examples using the exact design, the adaptive design averaged over 49 different TOQ, and the static design averaged over 20 approximate designs. Design adaptation overhead, which represents the time for running the ML-based design selector, is 30.5 ms, 93.9 ms, 164.6 ms, 148.6 ms, and 42.1 ms, for the five examples, respectively. Moreover, the five examples have a data processing time based on three selected designs per example of 50.90 s, 50.91 s, 51.10 s, 51.69 s, and 51.04 s, respectively. Thus, for these five examples, the design

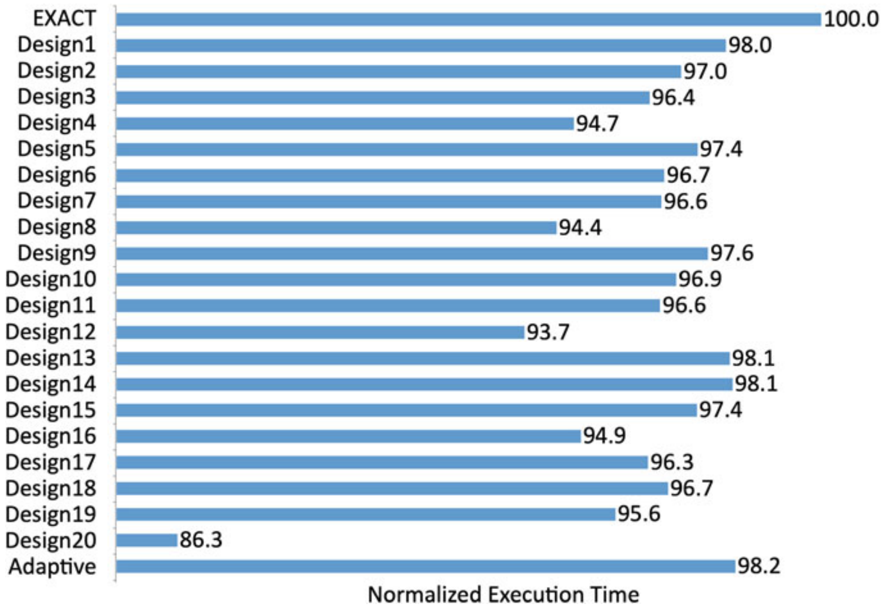

Fig. 6 Normalized execution time for image blending using 20 static designs

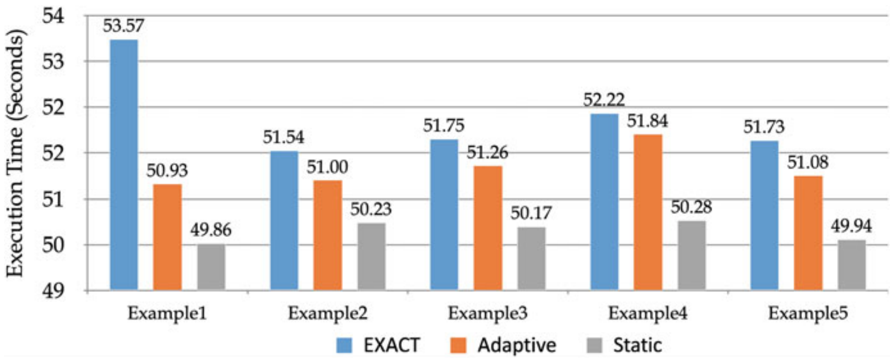

Fig. 7 Execution time of the exact, static, and adaptive design

adaptation time represents 0.06%, 0.18%, 0.32%, 0.28%, and 0.08% of the total execution time, respectively, which is a negligible overhead.

Energy Analysis of Adaptive Design Designing a library of approximate arithmetic modules aims to enhance the energy efficiency [13]. To calculate the energy consumed by the approximate multiplier to process an image, we use the following equation:

$$Energy = Power \times Delay \times N \tag{1}$$

Table 3 Obtained accuracy (PSNR) for various approximate designs

Application		KUL [20]	ETM [21]	ATCM [40]	Adaptive design (proposed)
Blending	*Set-1*, Ex. 1	24.8	27.9	41.5	61
	Set-1, Ex. 2	29.2	29.1	43.7	61.1
	Set-1, Ex. 3	20.3	24.8	33.1	63
	Set-1, Ex. 4	23	28	38.2	60.7
	Set-1, Ex. 5	27.6	29.4	40.3	61.5

where *Power* and *Delay* are obtained from the synthesis tool and N is the number of multiplications required to process an image, which equals $250 \times 400 = 10^5$ pixels. *Design9* multiplier has the highest energy consumption with 2970 pj and a saving of 896 pj compared to the exact design. Thus, the design adaption overhead of 733.7 pj is almost negligible compared to the total minimal energy savings of 89.6 µj (896 pj $\times 10^5$) obtained by processing a single image. These results validate our lightweight *design selector*.

Comparison with Related Work We now compare the output accuracy achieved by our adaptive design with the precision of two static approximate models based on approximate multipliers proposed by Kulkarni et al. [20] and Kyaw et al. [21] that have *similar structures* as the used approximate array multipliers. Moreover, we compare the accuracy of our work with a third approximate design based on the approximate tree compressor multiplier (ATCM), proposed by Yang et al. [40], which is a Wallace tree multiplier. Table 3 shows a summary of the obtained PSNR for image blending based on KUL [20], ETM [21], ATCM [40], and the proposed adaptive design. The proposed model achieves better output quality than static designs due to the ability to select the most suitable design from the approximate library.

3.4 Summary

For dynamic inputs, an approximate static design may lead to substantial output errors for changing data. Previous work has ignored the consideration of the changing inputs to assure the quality of individual outputs. We proposed a novel fine-grained input-dependent adaptive approximate design, based on machine learning models. Then, we implemented a fully automated toolchain utilizing a DT-based design selector. The proposed solution considers the inputs in generating the training data, building ML-based models, and then adapting the design to satisfy the *TOQ*. The "software" implementation of the proposed methodology, developed, provided a negligible delay overhead and was able to satisfy an output accuracy of 80% to 85.7% for image blending applications. Such quality-assured results come at the one-time cost of generating the training data and deploying and evaluating the design selector, i.e., a machine learning-based model. With runtime design adaptation, the model always identifies and selects the most suitable design for controlling the quality loss.

4 Hardware-Based Adaptive Design of Approximate Accelerators

The *software* implementation of the proposed adaptive approximate accelerate was able to satisfy the required TOQ with a minimum accuracy of 80%. Now, we present a hardware implementation of the adaptive approximate accelerator based on a field-programmable gate array (FPGA) and utilizing the feature of dynamic partial reconfiguration (DPR), with a database of 21 reconfigurable modules.

An essential advantage of FPGAs is their flexibility, where these devices can be configured and reconfigured on-site and at runtime by the user. In 1995, Xilinx introduced the concept of *partial reconfiguration* (PR) in its XC6200 series to increase the flexibility of FPGAs by enabling re-programming parts of design at runtime while the remaining parts continue operating without interruption [41]. The basic assumption of PR is that the device hardware resources can be time-multiplexed, similar to the ability of a microprocessor to switch tasks. PR eliminates the need to reconfigure and re-establish links fully and dramatically improves the flexibility that FPGAs offer. PR enables adaptive and self-repairing systems with reduced area and dynamic power consumption.

We propose to dynamically adapt the functionality of the FPGA-based approximate accelerators using machine learning (ML) and dynamic partial reconfiguration (DPR). We utilize the previously proposed DT- and NN-based *design selector* that continually monitors the input data and determines the most suitable approximate design and then, accordingly, partially reconfigures the FPGA with the chosen approximate design while maintaining the whole error-tolerant application intact. The proposed methodology applies to any error-tolerant application where we demonstrate its effectiveness using an image processing application. As FPGA vendors announced the technical support for the runtime partial reconfiguration, such systems are becoming feasible. To our best knowledge, the design framework for adaptively changeable approximate functional modules with input awareness does not exist.

4.1 Dynamic Partial Reconfiguration (DPR)

Field-programmable gate array (FPGA) devices conceptually consist of [42] (i) hardware logic (functional) layer which includes flip-flops, lookup tables (LUTs), block random-access memory (BRAM), digital signal processing (DSP) blocks, routing resources, and switch boxes to connect the hardware components and (ii)

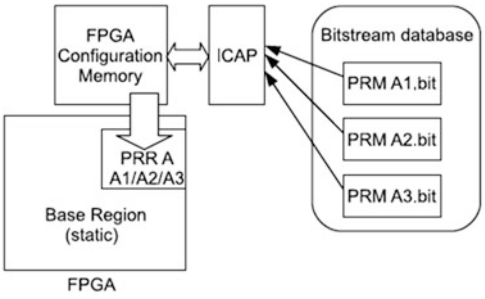

Fig. 8 Principle of dynamic partial reconfiguration on Xilinx FPGAs

configuration memory which stores the FPGA configuration information through a binary file called *configuration file* or *bitstream* (BIT). Changing the content of the *bitstream* file allows us to improve the functionality of the hardware logic layer. Xilinx and Intel (formerly Altera) are the leading manufacturing companies for FPGA devices. We use the VC707 evaluation board from Xilinx, which provides a hardware environment for developing and evaluating designs targeting the Virtex-7 XC7VX485T-2FFG1761C FPGA.

Partial reconfiguration (PR) is the ability to modify portions of the modern FPGA logic by downloading partial *bitstream* files while the remaining parts are not altered [43]. PR is a hierarchical and bottom-up approach and is an essential enabler for implementing *adaptive* systems. It can be static or dynamic, where the reconfiguration can occur while the FPGA logic is in the reset state or running state, respectively [42]. The DPR process consists of two phases: (i) fetching and storing the required bitstream files in the flash memory, which is not time-critical, and (ii) loading bitstreams into the reconfigurable region through a controller, i.e., internal configuration access port (ICAP). Implementing a partially reconfigurable FPGA design is similar to implementing multiple non-partial reconfiguration designs that share a common logic. Since the device is switching tasks in hardware, it has the benefit of both flexibility of software implementation and the performance of hardware implementation. However, it is not commonly employed in commercial applications [43].

Logically, the part that will host the reconfigurable modules (dynamic designs) is the dynamic *partial reconfigurable region* (PRR), which is shared among various modules at runtime through multiplexing. Figure 8 illustrates a reconfigurable design example on Xilinx FPGAs, with a partially reconfigurable region (PRR) A, which is reserved in the overall design layout mapped on the FPGA, with three possible partially reconfigurable modules (PRM). During PR, a portion of the FPGA needs to keep executing the required tasks, including the reconfiguration process. This part of the FPGA is known as the *static region*, which is configured only once at the boot time with a full *bitstream*. This region will also host static parts of the system, such as I/O ports as they can never be physically moved. When a hardware (signal) or a software (register write) trigger event occurs, the Partial Reconfiguration Controller (PRC) fetches/pulls partial *bitstreams* from the memory/database and delivers them to a configuration port.

4.2 Machine Learning-Based Models

Here, we describe the FPGA-based implementation of the design selector based on DT and NN models.

4.2.1 Decision Tree-Based Design Selector

As described previously, the DT-based models have an average accuracy of 77.8% and 74.3% for predicting the design *Degree* and *Type*, respectively, as shown in Table 1. Time overhead for executing the *software* implementation of these models is around 24.6 ms in total, with 8.87 ms to predict the design *Degree* and 15.72 ms to predict the design *Type*. This section evaluates the power, area, delay, and energy of the *FPGA-based* implementation of the DT-based *design selector*. We utilize the XC6VLX75T FPGA, which belongs to the Virtex-6 family. The configurable logic block (CLB) comprises 2 slices, each containing 4 6-input LUTs and 8 flip-flops, for a total of 8 6-input LUTs and 16 flip-flops per CLB. We use Mentor Graphics ModelSim [44] for functionality verification. We use Xilinx XPower Analyzer for the power calculation based on exhaustive design simulation [45], while for logic synthesis, we use the Xilinx Integrated Synthesis Environment (ISE 14.7) tool suite [46].

The obtained characteristics of the DT-based model are shown in Table 4, where the power consumption of the model ranges between 35 mW and 44 mW. This value is insignificant compared to the power consumption of approximate multipliers, where these multipliers being selected are used for *N* inputs. Similarly, the introduced area, delay, and energy overhead are amortized by running the approximate design for *N* inputs. The area of the model, represented in terms of the number of slice LUTs, is 1099, at maximum. Also, the number of occupied slices could reach 452 slices. The worst-case frequency that the model could run is 43.65 MHz, with a period of 22.91 ns. The designed model could consume a maximum energy of 733.7 pj.

The *design selector*, which is synthesized only once, is specific for the considered set of approximate designs. However, the proposed methodology is applicable to other approximate designs as well. The implementation overhead, i.e., power, area, delay, and energy, for the DT-based model is insignificant compared to the approximate accelerator since it is a simple nesting of if-else statements with a maximum depth of 12 to reach a node of a final result.

4.2.2 Neural Network-Based Design Selector

Neural networks (NNs) have typically been implemented in software. However, recently with the exploding number of embedded devices, the hardware implementation of NNs is gaining substantial attention. FPGA-based implementation of NN is complicated due to a large number of neurons and the calculation of complex equations such as activation function [47]. We use the sigmoid function $f(x)$ as an activation function. A piecewise second-order approximation scheme for the implementation of the sigmoid function is proposed in [48] as provided by Eq. (2). It has inexpensive hardware, i.e., one multiplication, no lookup table, and no addition.

$$f(x) = \begin{cases} 1 & x > 4.0 \\ 1 - \frac{1}{2}(1 - \frac{|x|}{4})^2, & 0 < x \leq 4.0 \\ \frac{1}{2}(1 - \frac{|x|}{4})^2, & -4.0 < x \leq 0 \\ 0, & x \leq -4.0 \end{cases} \tag{2}$$

As shown in Table 1, we implemented a two-step design selector by predicting the design *Degree* first and then the *Type*, with an accuracy of 82.17% and 67.3%, respectively. The execution time of the NN-based model ranges between 37.6 ms and 26.3 ms, with an average of 32.7 ms.

We implemented the NN-based model on FPGA, and its characteristics, including dynamic power consumption, slice LUTs, occupied slices, operating frequency, and consumed energy, are shown in Table 4. These values are insignificant when compared to the characteristics of approximate multipliers, where these multipliers are used for N inputs. However, compared to the DT-based model, the NN-based model has an execution time, which is 1.31× higher than the DT, while its average accuracy is almost 0.98× of the accuracy achieved by the DT-based model. Moreover, regarding other design metrics, including power, slice LUTs, occupied slices, period, and energy, the NN-based model has a value of 8.06×, 13.93×, 11.74×, 1.61×, and 6.8×, consecutively, compared with the DT-based model. Unexpectedly, the DT-based model is better than the NN-based model in all design characteristics, including accuracy and execution time.

4.3 Adaptive Design Methodology

Figure 9 shows the FPGA-based methodology for quality assurance of approximate computing through design adaptation, inspired by the general methodology shown in Fig. 1. In order to utilize the available resources of the FPGA and show the benefits of design approximation, we integrate 16 multipliers into an accelerator to be used altogether. Figure 10 shows the internal structure for the approximate accelerator with 16 multipliers. Each input, i.e., A_i and B_i where $16 \geq i \geq 1$, is 8-bit wide.

The implemented ML-based models (design selectors) are DT-based only, where model training is done once offline, i.e., off-FPGA. Then, the VHDL implementation of the obtained DT-based model, which is the output of the offline phase, is integrated as a functional module within the online phase of the FPGA-based

Table 4 Power, area, delay, frequency, and energy of DT- and NN-based design selectors

Model		Dynamic power (mW)		Slice LUTs		Occupied slices		Period (ns)		Frequency (MHz)		Energy (pJ)	
Inputs	Output	DT	NN	DT	NN	DT	NN	DT	NN	DT	NN	DT	NN
C1, C2, PSNR	Degree	16	155	602	7835	231	2683	22.910	31.504	43.65	31.74	366.6	4883.1
C1, C2, PSNR, s2=D1	Type	19	164	497	8427	189	2791	18.596	31.746	53.78	31.50	353.3	5206.3
C1, C2, PSNR, s2=D2	Type	23	153	449	6625	221	2309	15.962	31.718	62.65	31.53	367.1	4852.8
C1, C2, PSNR, s2=D3	Type	23	159	390	5360	149	1731	15.494	29.164	64.54	34.29	356.4	4637.0
C1, C2, PSNR, s2=D4	Type	28	170	298	4549	134	1420	11.838	28.678	84.47	34.87	331.5	4875.3

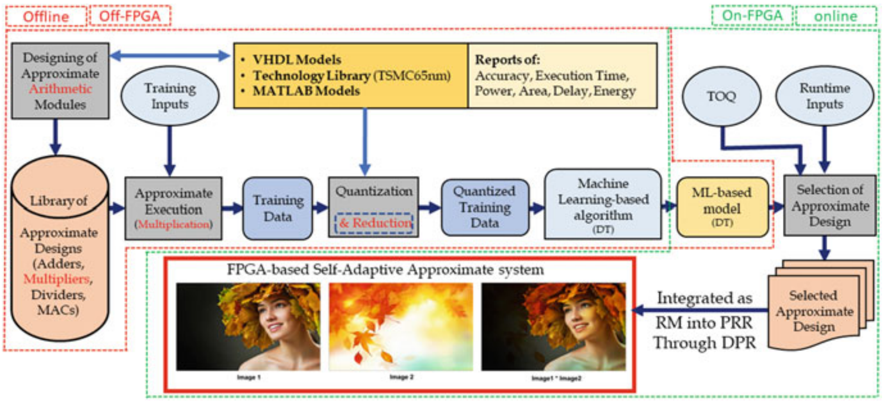
Offline
Off-FPGA
On-FPGA
online
Designing of Approximate Arithmetic Modules
Training Inputs
VHDL Models
Technology Library (TSMC65nm)
MATLAB Models
Reports of: Accuracy, Execution Time, Power, Area, Delay, Energy
TOQ
Runtime Inputs
Library of Approximate Designs (Adders, Multipliers, Dividers, MACs)
Approximate Execution (Multiplication)
Training Data
Quantization & Reduction
Quantized Training Data
Machine Learning-based algorithm (DT)
ML-based model (DT)
Selection of Approximate Design
FPGA-based Self-Adaptive Approximate system
Integrated as RM into PRR Through DPR
Selected Approximate Design

Fig. 9 Methodology of hardware-based adaptive approximate design

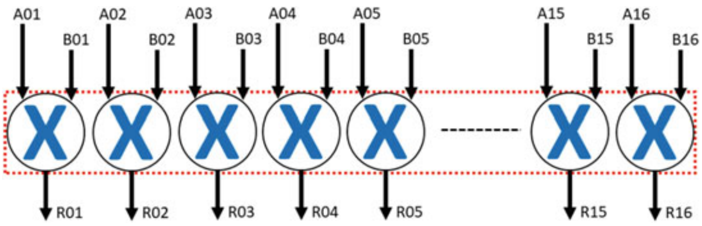

Fig. 10 An accelerator with 16 identical approximate multipliers

adaptive system, as shown in Fig. 11. The proposed FPGA architecture contains a set of intellectual property (IP) cores, connected through a standard bus interface. The developed approximate accelerator core is with the capability of adjusting processing features as commanded by the user to meet the given TOQ. For the parallel execution, we utilize the existing block RAM in the Xilinx 7 series FPGAs, which have 1030 blocks of 36Kbits. Thus, we store the input data (images) in a distributed memory, e.g., save each image of size 16 KByte into 16 memory slots each of 1 KByte. Other configurations of the memory are also possible and can be selected to match the performance of the processing elements within the accelerator.

The *online* phase of the adaptive design, based on the decision tree, is presented in Fig. 11, where the annotated numbers, i.e., ① to ⑧, show the flow of its execution for image blending application. The target device is xc7vx485tffg1761-2, and the evaluation kit is Xilinx Virtex-7 VC707 Platform [49]. The main components are the reconfiguration engine, i.e., DT-based design selector, and the reconfigurable core (RC), i.e., approximate accelerator. The RC is placed in a well-known partially reconfigurable region (PRR) within the programmable logic.

We evaluate the effectiveness of the proposed methodology for an FPGA-based adaptive approximate design utilizing DPR. For that, we select an *image blending* application due to its computationally intensive nature and its amenability

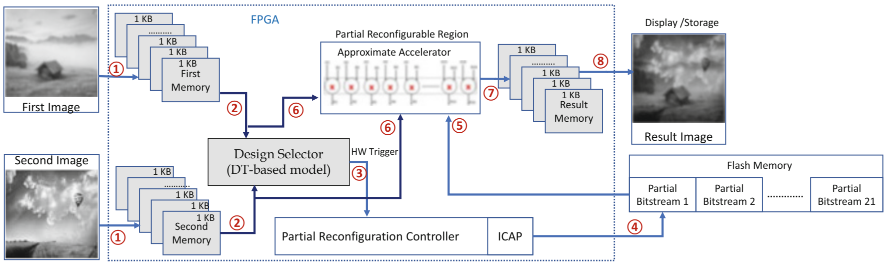

Fig. 11 Methodology of FPGA-based adaptive approximate design—online phase

to approximation. As a first step, to prove the validity of the proposed design adaptation methodology, we evaluate a design without the DPR feature, utilizing the exact accelerator as well as 20 approximate accelerators that exist simultaneously, based on the proposed methodology. Thus, 21 different accelerators evaluate the outputs. Next, based on the inputs and the given TOQ, the design selector chooses the output of a specific design, which has been selected based on the DT model. Finally, the selected result will be forwarded as the final result of the accelerator. The evaluated area and power consumption of such a design are 15× and 24× more significant than the exact implementation, respectively.

We use MATLAB to read the images, re-size them to 128 × 128 pixels, convert them to grayscale, and then write them into coefficient (.COE) files. Such files contain the image pixels in a format that the Xilinx CORE Generator can read and load. We store the images in an FPGA block RAM (BRAM). The design evaluates the average of the pixels of each image retrieved from the memory; then, the *hardware selector* decides which reconfigurable module, i.e., *bitstream* file, to load into the reconfigurable region. The full bitstream is stored in flash memory to be booted up into the FPGA at power-up. Moreover, the partial bitstreams are stored in well-known addresses of the flash memory.

4.4 Experimental Results

In the following, we discuss the results of our proposed methodology when evaluated on image processing applications. In particular, we present the obtained accuracy results along with reports of the area resources utilized by the implemented system.

Accuracy Analysis of the Adaptive Design We evaluate the accuracy of the proposed design over 55 examples of image blending. For each example, our TOQ (PSNR) ranges from 15 dB to 63 dB. The images we use are from the database of "8 Scene Categories Dataset" [50], which is downloadable from [51]. Figure 12 shows the minimum, maximum, and average curves of the obtained output quality,

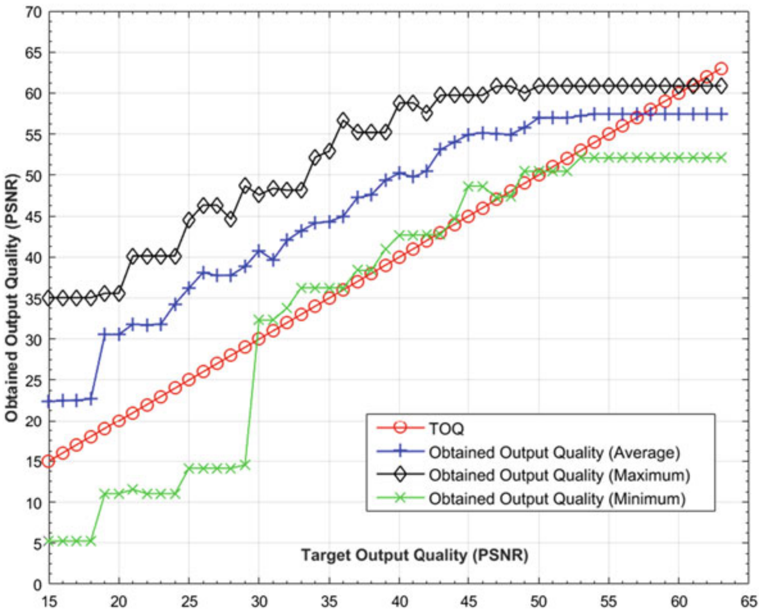

Fig. 12 Obtained output quality for FPGA-based adaptive image blending

each evaluated over 55 examples. Generally, for image processing applications, the quality is typically considered acceptable if PSNR is 30 dB at least and otherwise unacceptable [52]. Based on that, the design adaptation methodology has been executed 1870 times, while the TOQ has been satisfied 1530 times. Thus, the accuracy of our obtained results in Fig. 12 is 81.82%.

Area Analysis of the Adaptive Design Table 5 shows the primary resources of the XC7VX485T-2FFG1761 FPGA [53]. Moreover, it shows the resources required for the image blending application utilizing an approximate accelerator, both static and adaptive implementation. Design checkpoint files (.DCP) are a snapshot of a design at a specific point in the flow, which includes the current netlist, any optimizations made during implementation, design constraints, and implementation results. For the static implementation, the .DCP file is 430 KByte only, while for the dynamic implementation, it is 17411 KByte. This increase in the file size is due to the logic which has been added to enable DPR, as well as the 20 different implementations for the reconfigurable module (RM). Moreover, the overhead of such logic is shown in the increased number of occupied slice LUTs and slice registers. However, both static and dynamic implementations have the same size of the *bitstream* file (692 KByte), which is to be downloaded into the FPGA. DPR enables downloading the partial *bitstream* into the FPGA rather than the full *bitstream*. Thus, downloading 692 KByte rather than 19799 KByte would be 28.6 × faster. Since different variable-size reconfigurable modules will be assigned to the same reconfigurable region,

Table 5 Area/size of static and adaptive approximate accelerator

Design	.DCP file KByte	Slice LUTs	Slice registers	RAMB36	RAMB18	Bonded IOB	DSPs	Bitstream size (KByte)
XC7VX485T-2FFG1761 FPGA	–	303600	607200	1030	2060	700	2800	–
Static design	430	1472	357	235	51	65	0	19799
Adaptive–Top	17411	12876	15549	235	51	65	0	19799
Adaptive–Exact RM	770	1287	0	0	0	0	0	692
Adaptive–Max Approx RM	647	800	0	0	0	0	0	692
Adaptive–Min Approx RM	458	176	0	0	0	0	0	692

it must be large enough to fit the biggest one, i.e., the exact accelerator in our methodology.

Table 5 shows the main features of the Xilinx XC7VX485T-2FFG1761 device, including the number of slice LUTs, slice registers, and block RAM. The total capacity of block RAM is 37080 Kbit, which could be arranged as 1030 blocks of size 36Kbit each or 2060 blocks of size 18Kbit each. The reconfigurable module (RM) with exact implementation occupies 1287 slice LUTs. However, the number of slice LUTs occupied by the RM with approximate implementation varies from 800 to 176 LUTs. Thus, the area of the approximate RM varies from 62.16% to 13.68% of the area of the exact RM. Despite all of that, all 21 RMs have the same *bitstream* size of 692 KB.

4.5 Summary

To ensure the quality of approximation by design adaptation, we described the proposed methodology to adapt the architecture of the FPGA-based approximate design using dynamic partial reconfiguration. The proposed design with low power, reduced area, small delay, and high throughput is based on runtime adaptation for changing inputs. For this purpose, we utilized a lightweight and energy-efficient *design selector* built based on decision tree models. Such input-aware *design selector* determines the most suitable approximate architecture which satisfies user-given quality constraints for specific inputs. Then, the partial *bitstream* file of the selected design is downloaded into the FPGA. Dynamic partial reconfiguration allows quickly reconfiguring the FPGA devices without having to reset the complete device. The obtained analysis results of the image blending application showed that it is possible to satisfy the TOQ with an accuracy of 81.82%, utilizing a partial *bitstream* file that is 28.6$\times$ smaller than the full *bitstream*.

5 Conclusions

Approximate computing has re-emerged as an energy-efficient computing paradigm for error-tolerant applications. Thus, it is promising to be within the architecture and algorithms of brain-inspired computing, which has massive device parallelism and the ability to tolerate unreliable operations. However, there are essential questions to be answered before approximate computing can be made a viable solution for energy-efficient computing, such as [54] (1) how much to approximate at the component level to be able to observe the gains at the system level, (2) how to measure the final quality of approximation, and (3) how to maintain the desired output quality of an approximate application.

Toward addressing these challenges, we proposed a methodology that assures the quality of approximate computing through design adaptation based on fine-grained inputs and user preferences. For that, we designed a lightweight machine learning-based model, which functions as a design selector, to select the most suitable approximate designs to ensure the final quality of the approximation.

We proposed a novel methodology to generate an adaptive approximate design that satisfies user-given quality constraints, based on the applied inputs. For that, we have built a machine learning-based model (that functions as a design selector) to determine the most suitable approximate design for the applied inputs based on the associated error metrics. To solve the *design selector* model, we used decision tree and neural network techniques to select the approximate design that matches the closest accuracy for the applied inputs.

We realized the *software* and *hardware* implementations of the proposed methodology, with negligible overhead. The obtained analysis results of the image processing application showed that it is possible to satisfy the TOQ with accuracy ranging from 80% to 85.7% for various error-resilient applications. The *FPGA-based* adaptive approximate accelerator with constraints on size, cost, and power consumption relies on dynamic partial reconfiguration to assist in satisfying these requirements. In summary, the general proposed design adaptation methodology can be seen as a basis for automatic quality assurance. It offers a promising solution to reduce the approximation error while maintaining approximation benefits.

References

1. B. Moons, M. Verhelst, Energy-efficiency and accuracy of stochastic computing circuits in emerging technologies. IEEE J. Emerging Sel. Top. Circuits Syst. **4**(4), 475–486 (2014)

2. J. Han, M. Orshansky, Approximate computing: An emerging paradigm for energy-efficient design, in *European Test Symposium* (2013), pp. 1–6
3. S. Venkataramani, S.T. Chakradhar, K. Roy, A. Raghunathan, Approximate computing and the quest for computing efficiency, in *Design Automation Conference* (2015), pp. 1–6
4. R. Ragavan, B. Barrois, C. Killian, O. Sentieys, Pushing the limits of voltage over-scaling for error-resilient applications, in *Design, Automation Test in Europe* (2017), pp. 476–481
5. P. Roy, R. Ray, C. Wang, W.F. Wong, ASAC: automatic sensitivity analysis for approximate computing. SIGPLAN Not. **49**(5), 95–104 (2014)
6. V. Gupta, D. Mohapatra, A. Raghunathan, K. Roy, Low-power digital signal processing using approximate adders. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **32**(1), 124–137 (2013)
7. R. Nair, Big data needs approximate computing: technical perspective. Commun. ACM **58**(1), 104–104 (2014)
8. A. Mishra, R. Barik, S. Paul, iACT: A software-hardware framework for understanding the scope of approximate computing, in *Workshop on Approximate Computing Across the System Stack* (2014), pp. 1–6
9. J. Bornholt, T. Mytkowicz, K. McKinley, UnCertain: a first-order type for uncertain data. SIGPLAN Not. **49**(4), 51–66 (2014)
10. M. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, L. Tang, Input responsiveness: using canary inputs to dynamically steer approximation, in *Programming Language Design and Implementation* (ACM, New York, 2016), pp. 161–176
11. V.K. Chippa, S.T. Chakradhar, K. Roy, A. Raghunathan, Analysis and characterization of inherent application resilience for approximate computing, in *Design Automation Conference* (2013), pp. 1–9
12. E. Nogues, D. Menard, M. Pelcat, Algorithmic-level approximate computing applied to energy efficient hevc decoding. IEEE Trans. Emerg. Top. Comput. **7**(1), 5–17 (2019)
13. M. Masadeh, O. Hasan, S. Tahar, Comparative study of approximate multipliers, in *ACM Great Lakes Symposium on VLSI* (2018), pp. 415–418
14. M. Masadeh, O. Hasan, S. Tahar, Machine-learning-based self-tunable design of approximate computing. IEEE Trans. Very Large Scale Integr. VLSI Syst. **29**(4), 800–813 (2021)
15. H. Esmaeilzadeh, A. Sampson, L. Ceze, D. Burger, Neural acceleration for general-purpose approximate programs, in *International Symposium on Microarchitecture* (2012), pp. 449–460
16. M. Shafique, R. Hafiz, S. Rehman, W. El-Harouni, J. Henkel, Invited: cross-layer approximate computing: From logic to architectures, in *Design Automation Conference* (2016), pp. 1–6
17. S. Ullah, S. Rehman, B.S. Prabakaran, F. Kriebel, M.A. Hanif, M. Shafique, A. Kumar, Area-optimized low-latency approximate multipliers for FPGA-Based hardware accelerators, in *Design Automation Conference* (2018), pp. 1–6
18. M. Imani, R. Garcia, A. Huang, T. Rosing, Cade: configurable approximate divider for energy efficiency, in *Design, Automation Test in Europe Conference* (2019), pp. 586–589
19. G. Zervakis, K. Tsoumanis, S. Xydis, D. Soudris, K. Pekmestzi, Design-efficient approximate multiplication circuits through partial product perforation. IEEE Trans. Very Large Scale Integr. Syst. **24**(10), 3105–3117 (2016)
20. P. Kulkarni, P. Gupta, M. Ercegovac, Trading accuracy for power with an underdesigned multiplier architecture, in *International Conference on VLSI Design* (2011), pp. 346–351
21. K.Y. Kyaw, W.L. Goh, K.S. Yeo, Low-power high-speed multiplier for error-tolerant application, in *International Conference of Electron Devices and Solid-State Circuits* (2010), pp. 1–4
22. K.M. Reddy, Y.B.N. Kumar, D. Sharma, M.H. Vasantha, Low power, high speed error tolerant multiplier using approximate adders, in *VLSI Design and Test* (2015), pp. 1–6
23. M. Masadeh, O. Hasan, S. Tahar, Comparative study of approximate multipliers, in *Great Lakes Symposium on VLSI* (ACM, New York, 2018), pp. 415–418
24. M. Masadeh, O. Hasan, S. Tahar, Comparative study of approximate multipliers, in *CoRR*, vol. abs/1803.06587 (2018)
25. W. Baek, T. Chilimbi, Green: a framework for supporting energy-conscious programming using controlled approximation. SIGPLAN Not. **45**(6), 198–209 (2010)

26. M. Samadi, J. Lee, D. Jamshidi, A. Hormati, S. Mahlke, SAGE: self-tuning approximation for graphics engines, in *International Symposium on Microarchitecture* (2013), pp. 13–24
27. T. Wang, Q. Zhang, N. Kim, Q. Xu, On effective and efficient quality management for approximate computing, in *International Symposium on Low Power Electronics and Design* (2016), pp. 156–161
28. X. Chengwen, W. Xiangyu, Y. Wenqi, X. Qiang, J. Naifeng, L. Xiaoyao, J. Li, On quality trade-off control for approximate computing using iterative training, in *Design Automation Conference* (2017), pp. 1–6
29. M. Shafique, W. Ahmad, R. Hafiz, J. Henkel, A low latency generic accuracy configurable adder, in *Design Automation Conference* (ACM, New York, 2015), pp. 86:1–86:6
30. X. Sui, A. Lenharth, D. Fussell, K. Pingali, Proactive control of approximate programs, in *International Conference on ASPLOS* (ACM, New York, 2016), pp. 607–621
31. M. Masadeh, O. Hasan, S. Tahar, Input-conscious approximate multiply-accumulate (MAC) unit for energy-efficiency. IEEE Access **7**, 147129–147142 (2019)
32. D. Mohapatra, V.K. Chippa, A. Raghunathan, K. Roy, Design of voltage-scalable meta-functions for approximate computing, in *Design, Automation Test in Europe* (2011), pp. 1–6
33. S. Shalev-Shwartz, S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms* (Cambridge University Press, Cambridge, 2014)
34. M. Masadeh, O. Hasan, S. Tahar, Controlling approximate computing quality with machine learning techniques, in *Design, Automation and Test in Europe* (2019), pp. 1575–1578
35. L. Breiman, J. Friedman, R. Olshen, C. Stone, *Classification and Regression Trees* (Chapman and Hall, Wadsworth, 1984)
36. R.C. Barros, A.C. de Carvalho, A.A. Freitas, *Automatic Design of Decision-Tree Induction Algorithms* (Springer, Berlin, 2015)
37. A. Raha, V. Raghunathan, qLUT: Input-Aware quantized table lookup for energy-efficient approximate accelerators. ACM Trans. Embed. Comput. Syst. **16**(5s), 130:1–130:23 (2017)
38. S. Xu, B.C. Schafer, Approximate reconfigurable hardware accelerator: adapting the micro-architecture to dynamic workloads, in *International Conference on Computer Design* (IEEE, New York, 2017), pp. 113–120
39. M. Masadeh, O. Hasan, S. Tahar, Error analysis of approximate array multipliers, in *CoRR* (2019). https://arxiv.org/pdf/1908.01343.pdf
40. T. Yang, T. Ukezono, T. Sato, Low-power and high-speed approximate multiplier design with a tree compressor, in *International Conference on Computer Design* (2017), pp. 89–96
41. Partial Reconfiguration User Guide (2013). https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug702.pdf. Last accessed on 2023-02-24
42. K. Vipin, S.A. Fahmy, FPGA dynamic and partial reconfiguration: a survey of architectures, methods, and applications. ACM Comput. Surv. **51**(4), 72:1–72:39 (2018)
43. D. Koch, *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications* (Springer, Berlin, 2012)
44. Mentor Graphics Modelsim (2019). https://www.mentor.com/company/higher_ed/modelsim-student-edition. Last accessed on 2023-02-24
45. Xilinx XPower Analyser (2019). https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ug733.pdf. Last accessed on 2023-02-24
46. Xilinx Integrated Synthesis Environment (2019). https://www.xilinx.com/products/design-tools/ise-design-suite/ise-webpack.html. Last accessed on 2023-02-24
47. S. Ngah, R. Abu Bakar, A. Embong, S. Razali, Two-steps implementation of sigmoid function for artificial neural network in field programmable gate array. ARPN J. Eng. Appl. Sci. **11**(7), 4882–4888 (2016)
48. M. Zhang, S. Vassiliadis, J.G. Delgado-Frias, Sigmoid generators for neural computing using piecewise approximations. IEEE Trans. Comput. **45**(9), 1045–1049 (1996)
49. VC707 Evaluation Board for the Virtex-7 FPGA: User Guide (2019). https://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf. Last accessed on 2023-02-13

50. A. Oliva, A. Torralba, Modeling the shape of the scene: a holistic representation of the spatial envelope. Int. J. Comput. Vis. **42**(3), 145–175 (2001)
51. Modeling the shape of the scene: a holistic representation of the spatial envelope (2020). http://people.csail.mit.edu/torralba/code/spatialenvelope/. Last accessed on 2023-02-04
52. M. Barni, *Document and Image compression* (CRC Press, New York, 2006)
53. 7 Series FPGAs Data Sheet: Overview (2020). https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf. Last accessed on 2023-02-13
54. J. Han, Introduction to approximate computing, in *VLSI Test Symposium* (2016), pp. 1–1