A Quality-assured Approximate Hardware Accelerators–based on Machine Learning and Dynamic Partial Reconfiguration

MAHMOUD MASADEH, YASSMEEN ELDERHALLI, OSMAN HASAN, and SOFIENE TAHAR, Department of Electrical and Computer Engineering, Concordia University, Montreal, QC H3G 1M8, Canada

Machine learning is widely used these days to extract meaningful information out of the Zettabytes of sensors data collected daily. All applications require analyzing and understanding the data to identify trends, e.g., surveillance, exhibit some error tolerance. Approximate computing has emerged as an energy-efficient design paradigm aiming to take advantage of the intrinsic error resilience in a wide set of error-tolerant applications. Thus, inexact results could reduce power consumption, delay, area, and execution time. To increase the energy-efficiency of machine learning on FPGA, we consider approximation at the hardware level, e.g., approximate multipliers. However, errors in approximate computing heavily depend on the application, the applied inputs, and user preferences. However, dynamic partial reconfiguration has been introduced, as a key differentiating capability in recent FPGAs, to significantly reduce design area, power consumption, and reconfiguration time by adaptively changing a selective part of the FPGA design without interrupting the remaining system. Thus, integrating "Dynamic Partial Reconfiguration" (DPR) with "Approximate Computing" (AC) will significantly ameliorate the efficiency of FPGA-based design approximation. In this article, we propose hardware-efficient quality-controlled approximate accelerators, which are suitable to be implemented in FPGA-based machine learning algorithms as well as any error-resilient applications. Experimental results using three case studies of image blending, audio blending, and image filtering applications demonstrate that the proposed adaptive approximate accelerator satisfies the required quality with an accuracy of 81.82%, 80.4%, and 89.4%, respectively. On average, the partial *bitstream* was found to be 28.6× smaller than the full *bitstream*.

CCS Concepts: • **Computing methodologies → Machine learning algorithms**; **Classification and regression trees**; • **Hardware → Hardware accelerators**; **Reconfigurable logic and FPGAs;**

Additional Key Words and Phrases: Approximate computing, approximate hardware accelerator, decision tree, input-aware approximation, dynamic partial reconfiguration, adaptive design, FPGA

ACM Reference format:
Mahmoud Masadeh, Yassmeen Elderhalli, Osman Hasan, and Sofiene Tahar. 2021. A Quality-assured Approximate Hardware Accelerators–based on Machine Learning and Dynamic Partial Reconfiguration. *J. Emerg. Technol. Comput. Syst.* 17, 4, Article 57 (August 2021), 19 pages.
https://doi.org/10.1145/3462329

Authors' address: M. Masadeh, Y. Elderhalli, O. Hasan, and S. Tahar, Department of Electrical and Computer Engineering, Concordia University, Montreal, QC H3G 1M8, Canada.
Author's current address: M. Masadeh, Computer Engineering Department, Yarmouk University, Irbid 21163, Jordan.

1550-4832/2021/08-ART57 $15.00
https://doi.org/10.1145/3462329

1 INTRODUCTION

The class of brain-inspired **machine learning (ML)** algorithms that enable a computer to learn from observational data is called **Deep Neural Networks (DNNs)** [1]. DNNs are applicable in error-tolerant applications, such as video analysis, natural language processing, and image/speech recognition [2]. However, such large-scale networks have quite high memory and computational requirements. Thus, an energy-efficient realization of the DNNs is of paramount interest. For instance, Reference [3] explored novel models and topologies at the algorithmic level, while Reference [4] proposed emerging devices capable of mimicking the neuronal dynamics at the hardware level. Moreover, other brain-inspired machine learning approaches are also being implemented with emerging devices such as hyperdimensional computing [5], rhythmic spike [6], and high-dimensional computing [7].

Approximate computing (AC) has emerged, due to the breakdown of Moore's law and Dennard scaling, to enable processing the associated big-data with battery-powered systems in an energy- and resource-efficient manner. Another related important field, which has re-emerged is Stochastic Computing [8], which processes analog probabilities by means of digital circuits, and has recently been shown to provide an advantage for energy-efficient deep CNNs [9]. Judiciously approximating certain computations, such as addition [10] and multiplication [11], provides significant benefits in energy. AC is greatly suitable for applications, such as computer vision, data mining, scientific computing, machine learning, multimedia, and digital signal processing, that exhibit error-tolerance due to the following factors [12]: (i) redundant and noisy input data, (ii) lack of golden or unique output, (iii) imperfect perception in human sense where individual hearing and vision are not very sensitive, and (iv) implementation algorithms with self-healing and error attenuation patterns. **Neural networks (NN)** with intrinsic error-resiliency are capable of tolerating approximation in the underlying computations. Thus, approximation could be introduced at the hardware level of the DNN. Moreover, hardware approximation reduces accuracy could be recovered by re-training the network [13].

The fundamental computation unit in NN is a neuron, which performs a **multiply-accumulate (MAC)** operation. The most power-consuming operation among NN computation is multiplications. To address this issue, we propose using approximate multipliers. However, the workloads (input data) for these types of applications are increasingly dynamic [14] and thus can lead—for some inputs—to intolerable errors for a static approximate design, i.e., the design is fixed during the whole execution of the applied inputs. Therefore, to overcome this shortcoming, the approximate design should dynamically change for changing inputs. This is a critical requirement for convolution-based applications where the MAC operation consists about 90% of its computations [15]. Adapting the internal structure for a scalar/single approximate multiplier introduces area overhead for an FPGA-based design. Thus, it will not be addressed in this work. However, it is more suitable for embedded software design rather than FPGA-based design as shown in Reference [16].

FPGAs are usually very efficient in executing algorithms that involve simple arithmetic operations on a large amount of data [17]. Also, it is possible to adapt the implementation of a neural network such that it would be more beneficial when implementing it with FPGA such as **Bitwise Neural Network (BNN)** for resource-constrained environments [18, 19], integer approximation of the reservoir computing based on the mathematics of hyper-dimensional computing [20], and randomly connected neural network based on a resource-efficient fully-integer approach [21]. However, Reference [22] showed that despite the high performance offered by the **digital signal processing (DSP)** blocks, their usage might not be efficient in terms of overall performance and area requirements for some applications. An important advantage of FPGAs is their flexibility, where these devices can be configured and re-configured 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 a design at runtime while the remaining parts continue operating without interruption [23]. The basic premise of PR is that the device hardware resources can be time-multiplexed similar to the ability of a microprocessor to switch tasks. *Partial reconfiguration* eliminates the need to fully reconfigure and re-establish links, and dramatically enhances the flexibility that the FPGAs offer. This enables adaptive and self-repairing systems with reduced area and dynamic power consumption. Configuration at fine-granularity leads to higher reconfiguration time, while configuration at coarse-granularity results in resource wastage.

To control the quality of approximation and reduce the associated errors, various techniques have been proposed. However, none of these state-of-the-art techniques has explored the potential of different approximate hardware designs and their adaptation based on the applied inputs to improve the individual output quality. To address the above challenges, we propose to *dynamically adapt the functionality of the FPGA-based approximate accelerators using machine learning and dynamic partial reconfiguration.* The proposed technique integrates the benefits of machine learning, approximate computing, and dynamic partial reconfiguration and thus positively impacts the area and power consumption. In this article, we introduce a novel concept of a decision tree-based *design selector* that constantly monitors the input data and quickly determines the most suitable approximate design; then, accordingly partially reconfigures the FPGA with the selected approximate design while keeping the whole error-tolerant application intact. Although the proposed methodology is applicable to any error-tolerant application including neural networks and deep learning, we demonstrate its effectiveness using image and audio processing applications, which are based on a single multiplication operation per input data.

An approximate program is representable in a **data flow graph (DFG)**, which is a collection of nodes and their connecting edges, where a node represents an operation and the edge represents the connectivity of the nodes with their data flow. For the DFG of an approximate program it is required to identify the non-critical computations by analyzing the output sensitivity to the accuracy of the final results. Then, these computations can be executed approximately for the non-critical nodes while executing the critical nodes accurately. However, static approaches fail to consider the runtime information. To solve this problem, the authors of Reference [24] proposed a runtime approximate computing framework. They use a low-cost method to assess the effect of inputs on the accuracy of computation at every node in the DFG and then decide whether we should simply use approximate operation or perform an accurate computation. Our proposed work is tangential to Reference [24]. Thus, both approaches could be integrated together, where Reference [24] identifies if a specific multiplication operation is approximable or not based on the applied inputs. Then, our proposed solution picks the most suitable design to perform the specified operation. As a future work, we plan to extend our work by building an ML-based model to identify the approximable node/operation in a DFG, then another related model identifies the best suitable approximate design to use. In this work, we use three case studies that are based on a simple DFG with a single multiplication operation. Thus, we can avoid both under-approximation and over-approximation issues.

The main contributions of this article are the following: (1) a library of 20 FPGA-based approximate accelerators, which have a reduced area, power, delay, and energy compared to the exact design; (2) a lightweight decision tree-based design selector implemented on FPGA to pick the most suitable approximate design based on the applied inputs; (3) a runtime adaptive approximate system implemented on FPGA utilizing the feature of dynamic partial reconfiguration with a database of 21 reconfigurable modules.

The rest of the article is organized as follows: Section 2 introduces the related work to the assurance of quality of FPGA-based approximate designs. Section 3 explains the concept of dynamic partial reconfiguration. Section 4 discusses the proposed methodology and the design selector. Section 5 provides the system architecture and experimental results of an image blending application on adaptive designs. We conclude the article in Section 6 with some future work directions.

2 RELATED WORK

This section describes the state-of-the-art in quality assurance of approximate computing at both software and hardware levels. Then, it discusses their drawbacks and how our proposed work overcome these limitations. Approximate computing is applicable to all layers of the computing stack, i.e., software [25], architecture [26], and circuit [27]. A detailed description is surveyed in References [28, 29]. In References [30, 31], several research challenges related to this emerging computing paradigm have been described, including controlling the output error specially for dynamically changing inputs. Managing the quality of approximate hardware design for dynamically changing inputs has a significant importance to ensure 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, in this article, we aim to overcome the approximation error through an input-dependent design adaptation.

Mainly, there are two approaches for monitoring and controlling the accuracy of results of approximate systems at runtime. The first approach suggests to periodically, through *sampling techniques*, measure the error of an approximate system by comparing its result 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 system if the error is found to be above a pre-defined threshold, e.g., Green [32] and SAGE [33]. However, the quality of unchecked invocations cannot be ensured, and the previous quality violations cannot be compensated. The second approach relies on implementing a light-weight pre-trained error predictor to expect if the invocation of an approximate system would produce an unacceptable error for a particular input dataset [34–37]. However, the quality checker, proposed in Reference [34], is application-specific, and Reference [35] shows a low prediction accuracy for large applications. Wang et al. [36] proposed a quality management framework for AC, by using *multiple lightweight predictors*, that achieve a better energy efficiency than References [34] and [35] through avoiding unnecessary rollback recoveries. Xu et al. [37] proposed a framework supporting an iterative training process, to coordinate the training of the classifier and the accelerator with an intelligent selection of training data. However, the works reported in References [32–37], 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.

Raha et al. [38] designed a dual-mode, reconfigurable adder block. Similarly, four designs of 4-to-2 compressors are proposed in Reference [39]. Moreover, the deployment of a self-adaptive image filter, which is able to choose among different degrees of binary adder approximations at runtime, is also demonstrated in Reference [40]. However, the approaches, reported in References [38–40], either have very limited configuration options (e.g., Reference [38]), area overhead (e.g., Reference [39]), or latency overhead (e.g., Reference [40]). Recently, Xu et al. [41] proposed a runtime reconfigurable manager to select the most suitable approximate design based on the detected input data distribution. However, they utilize approximate accelerators designed at the behavioral level for various coarse-grained expected input data distributions. In addition, the proposed approximate circuits heavily depend on the training data used during the approximation process, where not all possible workload distributions can be precharacterized. Thus, the real workload may differ completely from the training one. Xu et al. [42] also presented a self-tunable runtime adaptive approximate architecture that is suitable for **application-specific integrated circuit (ASIC)** designs. However, the used approximation techniques are **variable-to-variable (V2V)** and **variable-to-constant (V2C)** optimization only. Overall, none of these state-of-the-art techniques, i.e., References [32–42], exploits the potential of different settings of the approximate system and their adaptations based on a user-specified quality constraint to ensure the accuracy of the individual outputs, which is the main idea proposed in this work. Our proposed work is complementary to References [41] and [42] in the sense that our designs are approximated independently of the applied inputs (unlike Reference [41]) and encompasses various simplifications (unlike Reference [42]).

An adaptive edge detection filter using **dynamic partial reconfiguration (DPR)** has been proposed in Reference [43]. The effectiveness of the DPR feature for edge detection applications is evaluated on the filter with different scenarios varying in size, complexity, and intensity of computation, where the resource utilization and timing are evaluated. In Reference [44], a design flow for image and signal processing **IP (Intellectual Property)** cores based on FPGA dynamic partial reconfiguration have been proposed. The benefits of dynamic partial reconfiguration for embedded vision applications have been quantified in Reference [45]. These benefits include area, power, delay and energy reduction. The works, reported in References [43–45], show the benefits of DPR. However, the used designs are exact rather than approximate ones.

In this article, we propose to use FPGA dynamic partial reconfiguration to adapt the architecture of approximate accelerators using a decision tree-based design selector. The selected approximate accelerator is expected to satisfy a given quality constraint for specific input data at runtime. *To the best of our knowledge, this is the first work that dynamically adapts the FPGA-based approximate system utilizing dynamic partial reconfiguration.*

3 DYNAMIC PARTIAL RECONFIGURATION

In this section, we briefly describe the FPGAs architecture that can utilize partial reconfiguration. Thereafter, we explain their main characteristics and components that enable them to build an adaptive design in more detail.

Field Programmable Gate Arrays: FPGA devices conceptually consist of [46]: (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) 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 change the functionality of the hardware logic layer. Xilinx and Intel (formerly Altera) are the main manufacturing companies for FPGA devices, which are used to implement algorithms and systems that would be difficult to efficiently realize in software. Due to the hardware available for implementation, in this work, 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 [47].

Dynamic and Partial Reconfiguration: PR is the ability to modify portions of the modern FPGA logic by downloading partial *bitstream* files, while the remaining portions are not altered [48]. PR is a hierarchical and bottom-up approach and is an important 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 [46].

DPR process consists of two main 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

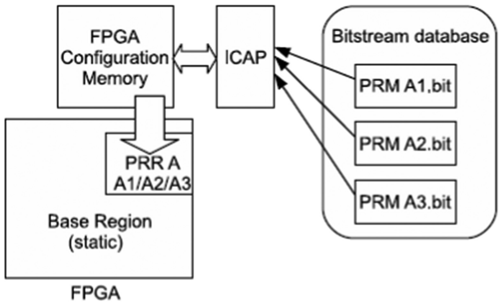

Fig. 1. Principle of dynamic partial reconfiguration.

region through a controller, i.e., **internal configuration access port (ICAP)**. Thus, DPR is valuable for devices that operate in a mission-critical environment that cannot be disrupted while some subsystems are being redefined. Implementing a partially reconfigurable FPGA design is similar to implementing multiple non-partial reconfiguration designs that share a common logic. Reconfigurable architectures are becoming increasingly popular due to their ability to adapt the architecture of the application [46]. Since the device is switching tasks in hardware, it has the benefit of both flexibility of a software implementation and the performance of a hardware implementation. However, it is not very widely employed in commercial applications yet compared to academia [48].

Benefits of Partial Reconfiguration: Adapting PR in FPGA design increases the effective logic density by time-multiplexing of hardware resources. A small *bitstream* file reduces the reconfiguration time as well as the external memory footprint, since PR files are smaller than the full configuration ones [46]. Therefore, real-time requirements can be met by using PR. Moreover, adaptive hardware systems, where computations can adapt to changing environments, can utilize PR. *Thus, the benefits of "Partial Reconfiguration" match the requirements of "Approximate Computing."* Therefore, our adaptive approximate accelerator with constraints on size, cost, and power consumption will rely on dynamic partial reconfiguration to satisfy these requirements.

Partial Reconfiguration Basic Model: Logically, the part that will host the reconfigurable modules is the dynamic ***partial reconfigurable region*** **(PRR)** that is shared among various modules at runtime through multiplexing, as shown in Figure 1. The shown model has a single PRR with three possible **partially reconfigurable modules (PRM)**. For FPGA full reconfiguration, all functional blocks should coexist while partial reconfiguration loads a single functional block at a time into the reconfigurable module. 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.

Xilinx Partial Reconfiguration Controller (PRC): It provides management functions for self-controlling partially reconfigurable designs [49]. It is designed for enclosed systems where all of the reconfigurable modules are known to the controller. The Xilinx PRC consists of multiple ***virtual socket*** **(VS)** managers, which connect to a single fetch path. The VS refers to a **re-configurable partition (RP)** with any supporting logic that exists in the static design. Each VS can contain up to 128 **re-configurable modules (RMs)** and up to 512 software and hardware triggers. In this work, our proposed design includes a single VS with 21 RMs each with a specific trigger. The mapping from a specific trigger, i.e., hardware or software, to a given RM is configurable during core configuration and at runtime. The PRC is not directly tied to any particular storage device where it fetches the *bitstream* data from an AXI4 bus [49], which allows the controller to access *bitstreams* no matter where they are stored.

Initially, the partial *bitstreams* are stored in a configuration library, i.e., *bitstream* database, as shown in Figure 1. When a hardware (signals) or a software (register write) trigger event occurs, the PRC fetches/pulls partial *bitstreams* from the memory/database through the AXI bus and delivers them to a configuration port, e.g., ICAP. For the PRC to quickly access the *bitstreams*, all of them should be available in a contiguous region of memory and indexed properly to allow locating the start address of an arbitrary *bitstream* efficiently without performing multiple look-ups.

High-speed Reconfiguration Controller: For Xilinx *SRAM-based* FPGAs, the ICAP is the basic component of any dynamic partial reconfigurable system [50]. The ICAP controller, which is *flexible* with high reconfiguration *throughput*, is responsible for executing all commands to access and modify the configuration memory [46]. The speed of configuration is directly related to the size of the partial *bitstream* file and the bandwidth of the configuration port. For ICAP, the data width is 32 bits and the bandwidth is 3.2 Gb/s. The maximum operational frequency of ICAP suggested by the vendor is 100 MHz, i.e., ICAP32@100 [51]. The ICAP enables DPR from within an FPGA chip, leading to the possibility of fully autonomous FPGA-based systems.

4 PROPOSED METHODOLOGY FOR FPGA-BASED ADAPTIVE APPROXIMATE SYSTEM

Performance enhancement of existing computing architectures is gradually saturating due to power and speed bottlenecks. Approximate computing is one of the promising techniques to enable new architectures. The preliminary results of AC show several orders of magnitude improvement for different metrics such as energy and area efficiency. However, the proposed approximate systems are of static nature despite their dynamically changing inputs, i.e., the output quality does not fluctuate and thus significantly violate the allowed error threshold for some inputs.

Figure 2 presents an overview of our novel generic design methodology that targets to resolve various challenges related to managing output quality of approximate computing [52], including: (i) minimizing the approximate results with large error magnitudes; (ii) selecting the approximate designs based on the input data as well as the user requirements; and (iii) continuously monitoring the output quality and compensating the error where monitoring all the inputs is not a feasible solution. Therefore, we propose a *lightweight* design selector based on computationally inexpensive ML-models, which are implementable in FPGAs [53]. The proposed methodology utilizes a **decision trees (DT)** algorithm to build an input-aware model, i.e., *AC quality manager*, to identify the most suitable approximate design based on the input data. Generally, DT-based models are known for their easy interpretation, where they use Boolean logic to construct a set of learning decision rules from the training data [54]. Moreover, decision trees are much smaller and easier to read and understand than other ML-based models, and they can easily be exported as a code that can be loaded into embedded systems [55].

Without loss of generality, we consider an adaptive approximate design with two knobs, i.e., *K1* and *K2*, that take values from the finite sets *s1* and *s2*, respectively, where *s1* represents the ***type*** of the approximate block used to build the approximate design, and *s2* is the approximation ***degree/level*** of the design. For example, we designed a set of energy-efficient approximate multipliers based on three design decisions [11]: (1) the type of the **full adders (FAs)** used to construct the multiplier where we used 11 different types; (2) the architecture, i.e., array or tree; and (3) the approximation degree (how much of the results to be approximated), where we have used two options: (i) all FAs are approximate (fully approximate design); and (ii) FAs that contribute to the least significant 50% of the resultant bits are approximated. Based on the obtained results, for this work, we have selected the most energy-efficient designs to be elements within the approximate

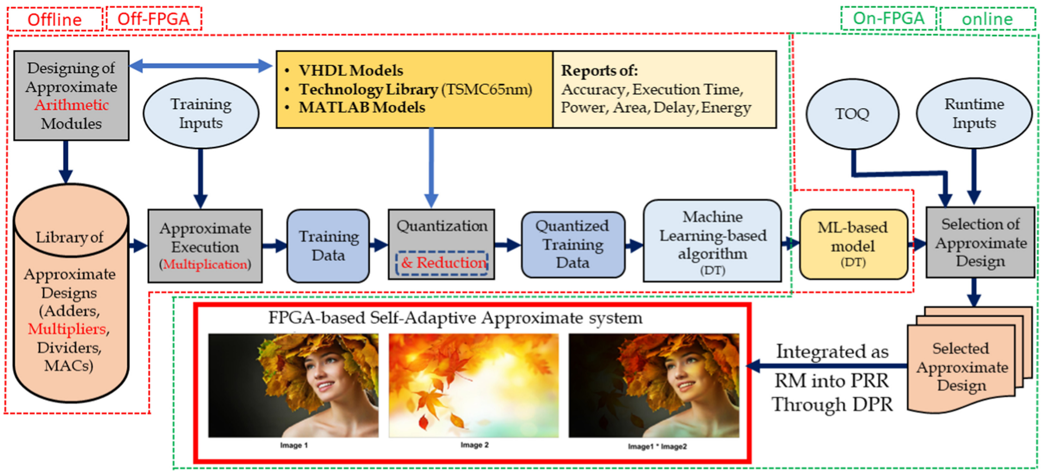
Offline
Off-FPGA
On-FPGA
online
Designing of Approximate Arithmetic Modules
Training Inputs
VHDL Models
Technology Library (TSMC65nm)
MATLAB Models
Reports of: Accuracy, Execution Time, Power, Area, Delay, Energy
TOQ
Runtime Inputs
Library of Approximate Designs (Adders, Multipliers, Dividers, MACs)
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
Image 1
Image 2
Image1 * Image2
Integrated as RM into PRR Through DPR
Selected Approximate Design

Fig. 2. Machine learning-based methodology for adaptive design to control the quality of approximate computing.

Table 1. Library of 20 Static Approximate Designs Based on *Degree* and *Type* Knobs [11]

Approximate Design		**Degree**			
		D1	**D2**	**D3**	**D4**
Type	**AMA1**	*Design1*	*Design2*	*Design3*	*Design4*
	AMA2	*Design5*	*Design6*	*Design7*	*Design8*
	AMA3	*Design9*	*Design10*	*Design11*	*Design12*
	AMA4	*Design13*	*Design14*	*Design15*	*Design16*
	AMA5	*Design17*	*Design18*	*Design19*	*Design20*

library, that are: (1) utilizing five types of full adders that are called approximate mirror adders, i.e., Type = $s1 = \{AMA1, AMA2, AMA3, AMA4, AMA5\}$, chosen from the low power approximate FAs [27], (2) its architecture is array, (3) the approximation degree has 4 options, i.e., Degree = $s2 = \{D1, D2, D3, D4\}$, where D1 has 7 bits approximated out of the 16-bit result, while D2, D3, and D4 have 8, 9, and 16 approximate bits, respectively. Table 1 shows our library of approximate designs based on *Degree* and *Type* knobs, i.e., $ApprxMul = \{Design1, \ldots\ldots, Design20\}$. Also, the library includes the exact design to be used whenever the required TOQ cannot be satisfied. The proposed design flow is adaptable, i.e., applicable to approximate functional units other than multipliers, e.g., approximate multiply-accumulate units [56] and approximate meta-functions [57].

As depicted in Figure 2, our proposed methodology encompasses two phases: (1) an offline phase (executed once), where the training inputs are applied to the library of approximate designs to generate the training data that is performed off-FPGAs, then we build an on-FPGA machine learning-based design selector; (2) an online phase, where the design selector continuously accepts the input data. Then, accordingly selects the most suitable design to match the required TOQ for the given inputs. Overall, the proposed methodology encompasses of the following main steps:

(1) Designing a Library of Approximate Arithmetic Modules: The first step is to design a library of approximate modules, e.g., approximate adder, multiplier, divider, square-root, and multiply-accumulate unit, which are suitable to be integrated as approximate accelerators. An approximate design with two knobs, i.e., *K1*, and *K2*, will have $|K1|x|K2|$ different design settings that constitute our library. For the example of an 8-bit approximate array multiplier, as shown in Table 1, we have $|K1|$ x $|K2| = 5 \times 4 = 20$ different settings with reduced power consumption, area, and execution time compared to the exact design. The error analysis of these design is available in Reference [58]. Moreover, using other state-of-the-art designs, such as Reference [59], is orthogonal to our approach and they can be included in a suitable way by implementing the necessary modifications in the library of approximate designs.

(2) Collecting of Training Data: For an approximate design with m inputs, each of n-bit width, and values ranging from 0 to 2^n-1, we apply j inputs to the approximate design to generate the training data, where $j \leq m$. For example, for an 8-bit approximate multiplier with $n = 8$, and to maximize models accuracy, we apply the inputs exhaustively where $j = m = 2$ and $n = 8$. Thus, we have 65,536 different input combinations. For 16 and 32-bit multipliers, the size of the input combinations is 4.29×10^9 and 1.86×10^{19}, respectively. Thus, rather a sampling of training data could be used due to the significantly large design space. For instance, in addition to the 8-bit version, we also apply an exhaustive simulation for 16-bit multipliers. Then, we pre-process (reduce) the training data based on designs characteristics, i.e., area, delay and power consumption [60].

(3) Clustering/Quantizing of Training Data: The accuracy of the proposed approximate library has been discussed in Reference [58]. Evaluating the design accuracy for a single input can provide the **error distance (ED)** metric only. Therefore, clustering of the training data is indispensable to evaluate the design accuracy over a range of consecutive inputs that belong to the same cluster. Thus, the mean error metrics, including **mean square error (MSE)** and **normalized mean error distance (NMED)**, are evaluated for each cluster. In this work, we consider the **Peak Signal-to-Noise Ratio (PSNR)**, which is a fidelity metric used to measure the quality of the output images, as given in Equation (1), as our TOQ. Clearly, PSNR depends on MSE, which is defined as the average of the squared ED values, as given in Equation (2) [61]. The model could be trained for various error metrics of approximate design such as ED and MSE. However, image statistical parameters such as entropy, mutual information, and edge preservation cannot be used in model building, since they are specific to images not to approximate designs (approximate multiplier in this work).

$$PSNR = 10 \times log_{10}\left(\frac{MAX^2}{MSE}\right) \tag{1}$$

$$MSE = \frac{1}{2^{2n}}\sum_{i=1}^{2^{2n}}|P_i - P_i'|^2 = \frac{1}{2^{2n}}\sum_{i=1}^{2^{2n}}|ED_i|^2 \tag{2}$$

For an 8-bit approximate multiplier, where each design has 65, 536 possible input combinations, we propose to cluster every 16 consecutive input values. Thus, each input encompasses 16 clustered inputs rather than 256. Therefore, the total number of possible input combinations per design is reduced to 256, and for the library of 8-bit approximate multiplier, we have a total number of $5 \times 4 \times 16^2 = 5,120$ training instances.

(4) Pre-processing/Reducing of Training Data: Data pre-processing is an often neglected but a major step in the data mining process [62]. However, in the proposed methodology with 8-bit designs, this step is not required as it is more applicable to designs with large training data. For the library of 16-bit approximate multipliers, we eliminate training instances that have a high **Area-Power-Delay-Product (APDP)** [63].

(5) Building an ML-based Design Selector: The design selector enables design adaptation for changing inputs to match the required TOQ where the error distribution is input-dependent [64]. ML-based algorithms find solutions by learning through a training data [53]. In supervised learning, a map between a set of input attributes and an output variable is used to predict the unseen data. The DT algorithm uses a flowchart-like tree structure to partition a set of data into various predefined classes, thereby providing the description, categorization, and generalization of the given datasets [63, 65]. We assume that only testing, as opposed to training, of a DT is done on an FPGA.

For 8-bit designs, we implemented the design selector utilizing the C5.0 function [66] in the statistical and computing R language [67]. The data was divided into two groups: the training-testing dataset and the validation dataset. 70% of the original dataset were selected randomly for model training using repeated 5-fold cross-validation. The most accurate model was selected for the testing stage based on 15% for the dataset. Finally, we validated the model using the remaining 15% for the dataset. The obtained accuracy depends on the settings of the different parameters, where the parameters of the most accurate DT-based models in R are as follows: (i) the maximum depth of any node of the tree (*maxdepth*) is 30; (ii) the minimum number of observations that must exist in a node for a split to be attempted (*minsplit*) is 20; and (iii) the minimum number of observations in any terminal node (*minbucket*) is 7. The implementation overhead, i.e., power, area, delay, and energy, for the DT-based model is negligible compared to the approximate accelerator, since it is a simple nesting of **if-else** statements. Moreover, for 16-bit designs, we use MATLAB's Classification Learner Toolbox [68] to build a decision tree-based model with an accuracy of 77.8%. Then, based on MATLAB's HDL Coder toolbox [69], we generate a portable and synthesizable VHDL code from MATLAB.

We utilized DTs and NNs to build an input-aware model, i.e., design selector, to pick the most suitable approximate design based on the input data. We implemented both the DT and NN-based model on FPGA, and evaluated their dynamic power consumption, slice LUTs, occupied slices, operating frequency, and consumed energy. These values are insignificant when compared to the characteristics of approximate multipliers, where these multipliers are used for 16,384 inputs. However, the NN-based model has an execution time that is 1.31× higher than the DT, while its average accuracy is almost 0.98× of the accuracy of the DT-based model. Moreover, the power, occupied slices, and energy of the NN-based model are 8.6×, 11.74×, and 13.6×, consecutively, compared with the DT-based model. Accordingly, we discarded the NN-based design selector due to the absence of advantages over DT.

(6) Integrating the Approximate Accelerator into Error Tolerant Application: The selected approximate accelerator is adapted within an error-resilient application. For example, accelerators based on 8-bit approximate multipliers are used in multiplication-intensive applications such as image processing, i.e., blending and filtering. Moreover, 16-bit approximate designs are used for audio blending. The online phase of the proposed methodology is implemented using Xilinx Vivado 2019.2, which supports dynamic partial reconfiguration feature [70]. To have a full control over each step of the design flow, we use TCL commands and scripts to manage design sources and processes in Non-Project Mode as proposed by Reference [71].

5 RESULTS AND DISCUSSION

This section evaluates the effectiveness of the FPGA-based implementation of the fully automated proposed system, including the approximate library and the DT-based design selector. We evaluate the proposed methodology based on two applications of image processing: (1) Image blending, where we use 55 examples; and (2) Image filtering, where we use two images, i.e., *Lina* and *Cameraman*. Moreover, we evaluate an audio mixing application with 45 examples, based on 16-bit models.

System Architecture: The proposed FPGA architecture contains a set of IP cores, e.g., AXI_EMC, JTAG_AXI, and ICAPE2, connected through a common bus interface. The developed approximate accelerator core is with the capability of adjusting processing features as commanded

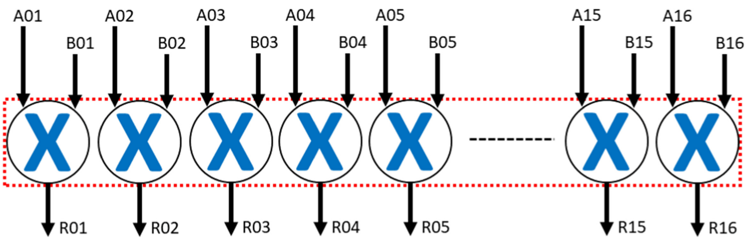

Fig. 3. An accelerator with 16 identical approximate multipliers.

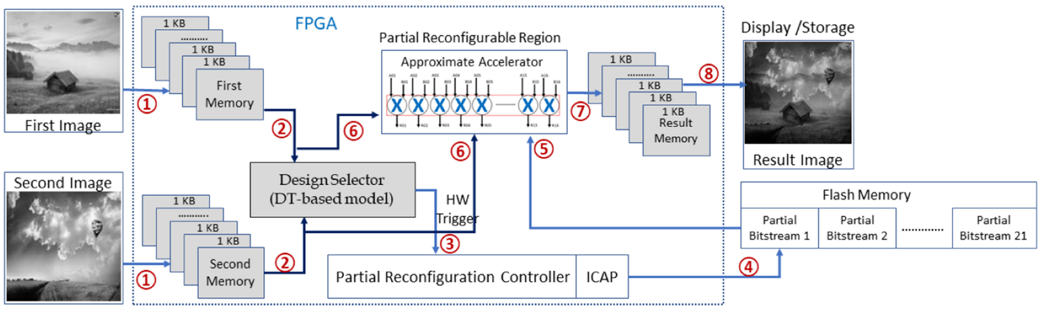

Fig. 4. The proposed self-adaptive approximate system utilizing decision tree model.

by the user to meet the given TOQ. Figure 3 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 width. For the parallel execution, we utilize the existing block RAM in Xilinx 7 series FPGAs, which have 1,030 blocks of 36 Kbits. Thus, we store the input data (image) in a distributed memory, e.g., store each image of size 16 KBytes 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 proposed self-adaptive approximate accelerator system based on decision tree is presented in Figure 4, where the annotated numbers show the flow of its execution. The target device is *xc7vx485tffg1761-2* and the evaluation kit is Xilinx Virtex-7 VC707 Platform [47]. 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. The AXI-HWICAP controller establishes communication with the ICAP.

We use MATLAB to read the images, re-size them to 128×128 pixels, convert them to grayscale, and then write 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)** that is considered as a special memory module separated from the regular logic cells in an FPGA [72]. 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.

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 to approximation. As a first step, to prove

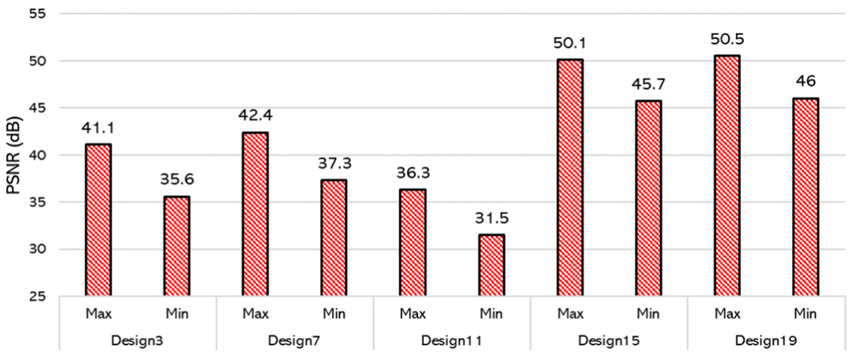

Fig. 5. Obtained output quality (PSNR) for static image blending.

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, a 21 different accelerators evaluate the outputs. Next, based on the inputs and the given TOQ, the design selector choose the output of a specific design, which have 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 design is 15× and 24× greater than the exact implementation, respectively.

Accuracy Analysis of a Static Design: For any approximate design, the obtained output quality depends on the value of the applied inputs [73]. Thus, for a static approximate accelerator, we apply image blending for different input data (images), where, accordingly, we obtain a variable PSNR. Figure 5 shows the maximum and the minimum of the PSNR that is obtained by applying different input images for a specific static design. The obtained PSNR fluctuates by 15.4%, 13.7%, 15.2%, 9.6%, and 9.8% for Design3, Design7, Design11, Design15, and Design19, respectively. Thus, for any design, the PSNR fluctuates for different images due to the dependency of the output quality on the inputs. Therefore, such high input-dependent achieved quality requires design adaptation to keep it minimal.

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 $15dB$ to $63dB$. The images we use are from the database of “8 Scene Categories Dataset” [74], which is downloadable from Reference [75]. It contains eight outdoor scene categories: coast, mountain, forest, open country, street, inside city, tall buildings, and highways. Figure 6 shows the minimum, maximum, and average curves of the obtained output quality, each evaluated over 55 examples. Generally, for image processing applications, the quality is typically considered acceptable if $PSNR \geq 30dB$, and otherwise unacceptable [76]. Based on that, the design adaptation methodology has been executed 1,870 times while the TOQ has been satisfied 1,530 times. Thus, the accuracy of our obtained results in Figure 6 is 81.82%.

Area Analysis of the Adaptive Design: Table 2 shows the main resources of the *XC7VX485T-2FFG1761* FPGA [77]. Moreover, it shows the resources required for the image blending application utilizing an approximate accelerator, both static and adaptive implementation. Design checkpoint files (.DCP) is a snapshot of a design at a specific point in the flow, which includes current netlist, any optimizations made during implementation, design constraints, and implementation results. For the static implementation the .dcp file is 430 KBytes only, while for the dynamic implementation it is 17,411 KBytes. This increase in the file size is due to the logic that has been added to enable dynamic partial reconfiguration, as well as the 20 different implementations for the **reconfigurable module (RM)**. Moreover, the overhead of such logic is shown in the increased

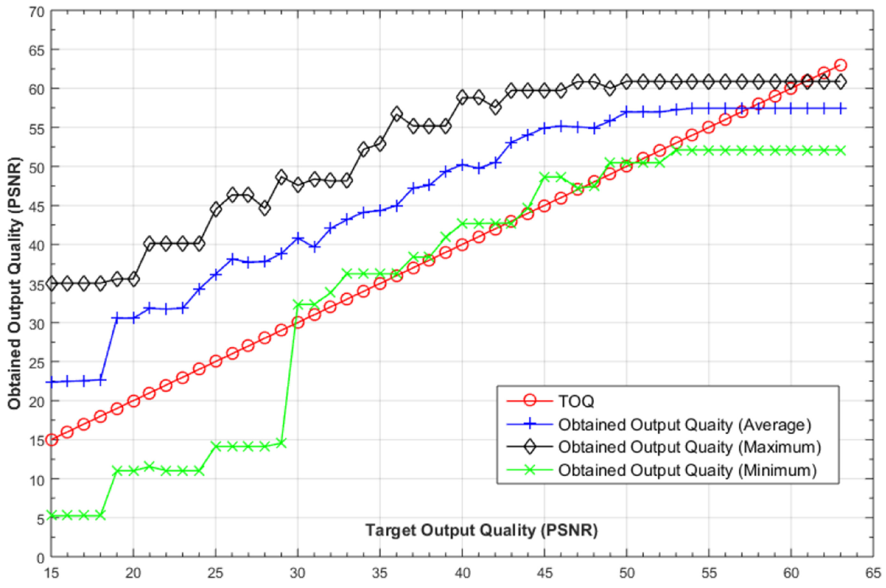

Fig. 6. Obtained output quality (PSNR) for FPGA-based adaptive image blending.

Table 2. Area/Size of Static and Adaptive Approximate Accelerator

Design	.DCP File	Slice LUTs	Slice Registers	RAMB36	RAMB18	Bonded IOB	DSPs	Bitsream size
XC7VX485T-2FFG1761 FPGA	–	303,600	607,200	1,030	2,060	700	2,800	–
Static Design	430 KBytes	1,472	357	235	51	65	0	19,799 KBytes
Adaptive - Top	17,411 KBytes	12,876	15,549	235	51	65	0	19,799 KBytes
Adaptive - Exact RM	770 KBytes	1,287	0	0	0	0	0	692 KBytes
Adaptive - Max Approx RM	647 KBytes	800	0	0	0	0	0	692 KBytes
Adaptive - Min Approx RM	458 KBytes	176	0	0	0	0	0	692 KBytes

number of the occupied Slice LUTs and Slice registers. However, both static and dynamic implementations have the same size of *bitstream* file (692 KBytes), 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 KBytes rather than 19,799 KBytes would be 28.6× faster. Since different variable-size reconfigurable modules will be assigned to the same reconfigurable region, it must be large enough to fit the biggest one, i.e., exact accelerator in our methodology.

Table 2 shows the main features for xilinx *XC7VX485T-2FFG1761* device including the number of slice LUTs, slice registers, and the number of block RAM. The total capacity of block RAM is 37,080 Kbits, which could be arranged as 1,030 blocks of size 36 Kbits each or 2,060 blocks of size 18 Kbits each. The **reconfigurable module (RM)** with exact implementation occupies 1,287 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 the 21 RMs have the same *bitstream* size, which is 692 KB.

Execution/Reconfiguration Time Analysis of the Adaptive Design: Current FPGA platforms are increasingly being used as final products rather than just rapid prototyping systems. However, the reconfiguration time for small bitstreams in partially reconfigurable FPGAs cannot be neglected. For instance, the authors of Reference [78] analyzed the system factors that affect the reconfiguration overhead. The total reconfiguration overhead depends on the system components, including the internal and external memory, the reconfiguration controller, the connections methods, the internal configuration port, and the FPGA configuration memory [78]. Modern FPGAs need several milliseconds or even seconds, depending on the size of the bitstreams, the configuration interface, and how these are stored and transferred [46].

The reconfiguration time of a partial bitstream is much smaller than the reconfiguration time of a full bitstream file. Thus, an important decision in partial reconfiguration is determining the *reconfiguration granularity*, which specifies the amount of computation to perform before design reconfiguration. In this work, design reconfiguration is done for each grayscale image of size 128 × 128 pixels, i.e., 16,384 pixels, where a single multiplication operation is done for each pixel. In our proposed methodology, the dynamic region is reconfigured if the TOQ or the inputs change.

$$Reconfiguration\ Time\ (RT) = \frac{Bitstream\ Size\ (BS)}{Reconfiguration\ Throughput\ (RTP)} \tag{3}$$

According to Equation (3), the reconfiguration time for a partial bitstream is 1.73 ms, while the reconfiguration time for the full bitstream is 49.5 ms. Moreover, the execution time for blending a single image of size 128 × 128 pixels is 163.83 μs. Generally, we should have a low overhead time compared to the execution time. Thus, as a future work, blending a video with 120 FPS (frames per second) for one second would have an 8% time overhead. Similarly, blending a video with 240 FPS for one second would have an 4% time overhead.

Energy Analysis of the Adaptive Design: The library of the approximate designs given in Table 1 has a reduced energy consumption [11]. The energy required to process an image is given by the following equation:

$$Energy = Power \times Delay \times N, \tag{4}$$

where *Power* and *Delay* are obtained from synthesizing the approximate multipliers, and *N* is the number of multiplications required to process an image, which equals 128 × 128=16,384 pixels. The energy consumption of the static design is 3,867 *pj*, while the energy consumption of the approximate design varies from 22 *pj* to 2,970 *pj*. Thus, the lowest energy saving is 897 *pj* per multiplication operation and 14.7 μj per image blending operation. Therefore, the overhead of energy consumption due to design adaptation of 734 *pj* is almost negligible compared to 14.7 μj.

Now, after we explained the proposed system for image blending application, we evaluate the accuracy of the proposed adaptive design for 16-bit audio blending and 8-bit image filtering applications.

Accuracy Analysis of the Adaptive Design for 16-bit Audio Blending: For sounds, which are propagating waves represented in a binary coding, we use the 16-bit depth format to be able to cover a wide range of amplitudes with an enhanced quality. Here, we implement a set of audio blending applications to evaluate the proposed design of adaptive approximation over 45 examples, where the TOQ (PSNR) ranges between 15 dB and 70 dB. The used WAV sound files were obtained from [79]. Figure 7 shows the obtained TOQ (PSNR) based on the adaptive design. The accuracy of the "average obtained TOQ," represented in a black curve, is 80.4%.

Accuracy Analysis of the Adaptive Design for 8-bit Image Filtering: Here, a Gaussian smoothing low-pass filter is used for accuracy evaluation of the adaptive system. The low-pass filter attenuates high frequency signals. Thus, image details are reduced. Gaussian smoothing includes convolving the image with a 2-D Gaussian function, according to Equation (5) [80]. Equation (6) shows the (3 × 3) kernel that we use, based on $\sigma = 1.5$, where the value of the central pixels significantly contributes to the kernel average.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{5}$$

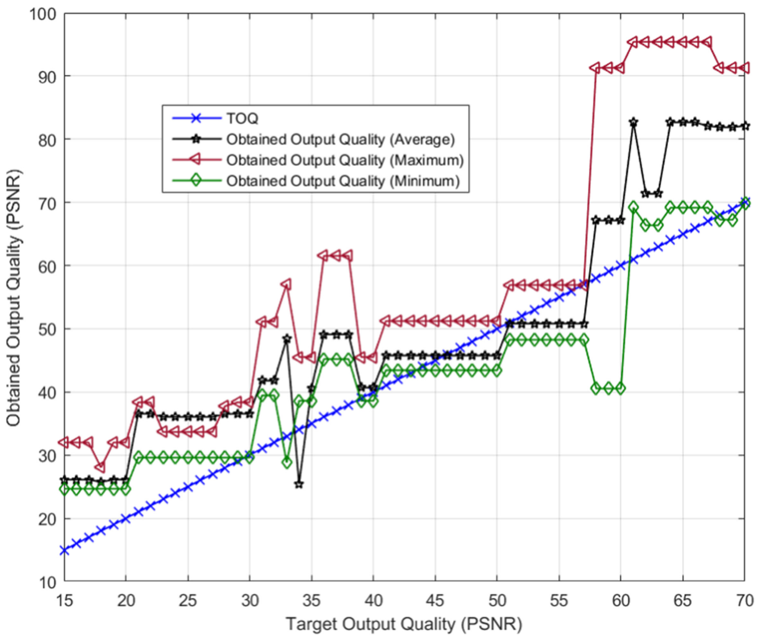

Fig. 7. Obtained output quality (PSNR) for FPGA-based adaptive audio blending.

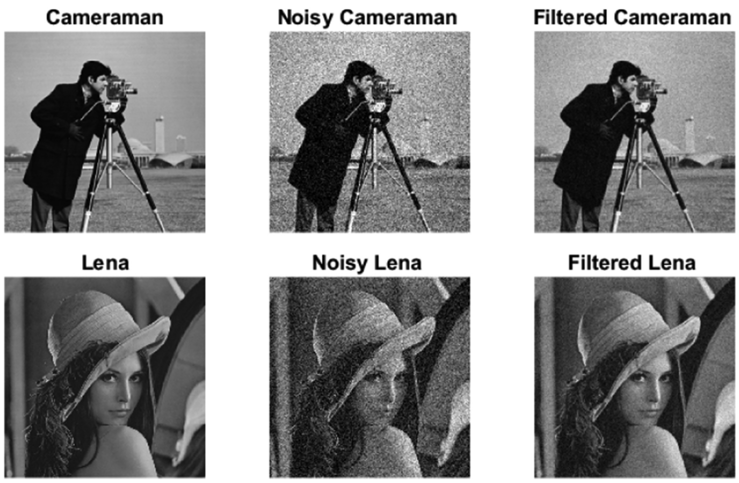

Fig. 8. *Cameraman* and *Lena* benchmarks.

$$Kernel = \frac{1}{254}\begin{bmatrix} 24 & 30 & 24 \\ 30 & 38 & 30 \\ 24 & 30 & 24 \end{bmatrix} \tag{6}$$

We used the benchmark images of *Cameraman* and *lena*, which are 8-bit gray-scale of size (128×128) pixels. The benchmark images are shown in Figure 8, where the Gaussian-noise added and the noisy images filtered with the exact design. For TOQ (PSNR) varying between 17 dB and 63 dB, Figure 9 shows the obtained PSNR for the benchmarks (*Cameraman* and *lena*) of image

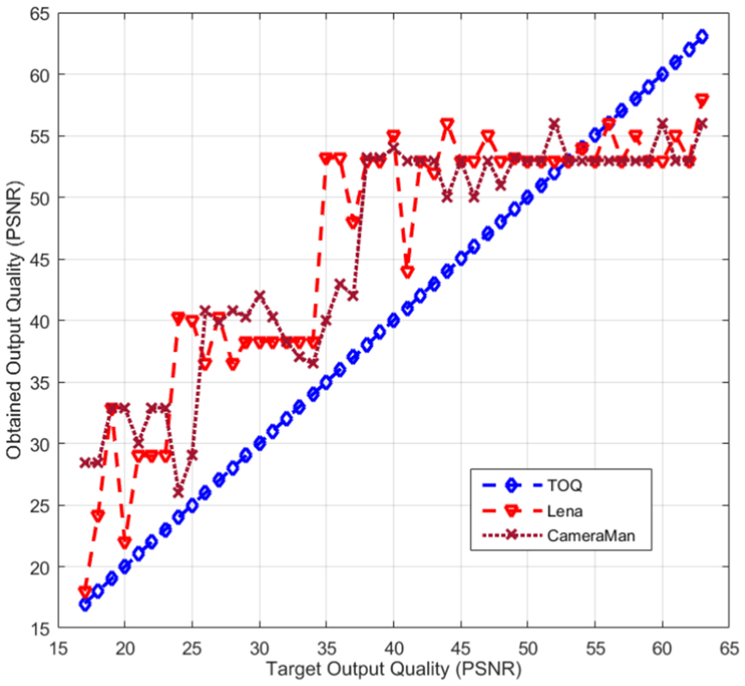

Fig. 9. Obtained output quality (PSNR) for FPGA-based adaptive image filtering.

filtering based on the adaptive design. Clearly, we notice that the adaptive system is always able to satisfy the TOQ for PSNR ≤ 53 dB, which is the best output quality obtained by static approximate designs and cannot be surpassed by the adaptive design. For the obtained results of 8-bit image blending and filtering, various characteristics can be evaluated, such as entropy and edge preservation. However, we cannot compare them with a reference value to evaluate the accuracy of the model. Thus, in this work, we only used PSNR. As future work, we could do the training at the image level.

6 CONCLUSION

Design approximation at the hardware level, e.g., approximate multipliers, are suitable to increase the energy-efficiency of FPGA-based error-tolerant applications including machine learning. However, it is clear that the compelling principle of approximate computing can lead to large output errors for dynamically changing inputs that are applied to static approximate systems. Therefore, in this article, we proposed to adapt the architecture of the PPGA-based approximate accelerator 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 that satisfies user-given quality constraints for specific inputs. Then, the partial *bitstream* file of the selected design is downloaded into the FPGA. This allows the quick swapping of modules in and out of the devices without having to reset the complete device. The obtained results for image blending, audio blending, and image filtering applications show that the *proposed adaptive design* is satisfying the user given TOQ for 81.82%, 80.4%, and 89.4% of the time, respectively, with a partial *bitstream* file that is 28.6× smaller than the full *bitstream*. As future work, we plan to target approximate accelerators based on arithmetic units rather than multipliers. Moreover, other machine learning algorithms would be investigated for higher accuracy and lower implementation overhead. For different error-resilient applications, we aim to reflect application-specific parameters by mapping them into the most suitable corresponding accuracy metrics.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.
[2] G. Dede and M. H. Sazlı. 2010. Speech recognition with artificial neural networks. *Dig. Sig. Process.* 20, 3 (2010), 763–768.
[3] R. Hoang, D. Tanna, L. Jayet Bray, S. Dascalu, and F. Harris. 2013. A novel CPU/GPU simulation environment for large-scale biologically realistic neural modeling. *Front. Neuroinform.* 7 (2013), 19.
[4] C. David Wright, P. Hosseini, and J. A. V. Diosdado. 2013. Beyond von-Neumann computing with nanoscale phase-change memory devices. *Adv. Funct. Mater.* 23, 18 (2013), 2248–2254.
[5] G. Karunaratne, M. L. Gallo, G. Cherubini, L. Benini, Abbas Rahimi, and A. Sebastian. 2019. In-memory hyperdimensional computing. *Nat. Electron.* 3 (2019), 327–337.
[6] E. Paxon Frady and Friedrich T. Sommer. 2019. Robust computation with rhythmic spike patterns. *Nat. Acad. Sci.* 116, 36 (2019), 18050–18059.
[7] A. Rahimi, S. Datta, D. Kleyko, E. P. Frady, B. Olshausen, P. Kanerva, and J. M. Rabaey. 2017. High-dimensional computing as a nanoscalable paradigm. *IEEE Trans. Circ. Syst. I: Reg. Pap.* 64, 9 (2017), 2508–2521.
[8] A. Alaghi and J. P. Hayes. 2018. Computing with randomness. *IEEE Spect.* 55, 3 (2018), 46–51.
[9] Vincent T. Lee, Armin Alaghi, John P. Hayes, Visvesh Sathe, and Luis Ceze. 2017. Energy-efficient hybrid stochastic-binary neural networks for near-sensor computing. In *Design, Automation and Test in Europe Conference.* 13–18.
[10] H. Jiang, J. Han, and F. Lombardi. 2015. A comparative review and evaluation of approximate adders. In *Great Lakes Symposium on VLSI.* ACM, 343–348.
[11] M. Masadeh, O. Hasan, and S. Tahar. 2018. Comparative study of approximate multipliers. In *Great Lakes Symposium on VLSI.* ACM, 415–418.
[12] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan. 2015. Approximate computing and the quest for computing efficiency. In *Design Automation Conference.* 1–6.
[13] S. S. Sarwar, S. Venkataramani, A. Raghunathan, and K. Roy. 2016. Multiplier-less artificial neurons exploiting error resiliency for energy-efficient neural computing. In *Design, Automation Test in Europe Conference.* 145–150.
[14] W. Quan and A. D. Pimentel. 2016. Scenario-based run-time adaptive MPSoC systems. *J. Syst. Archit.* 62 (2016), 12–23.
[15] H. Nakahara and T. Sasao. 2015. A deep convolutional neural network based on nested residue number system. In *International Conference on Field Programmable Logic and Applications.* 1–6.
[16] M. Masadeh, O. Hasan, and S. Tahar. 2021. Machine-learning-based self-tunable design of approximate computing. *IEEE Trans. Very Large Scale Integ. Syst.* 29, 4 (2021), 800–813.
[17] M. C. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello. 2007. Achieving high performance with FPGA-based computing. *Computer* 40, 3 (2007), 50–57.
[18] M. Kim and P. Smaragdis. 2016. Bitwise Neural Networks. arXiv:cs.LG/1601.06071
[19] M. Kim and P. Smaragdis. 2018. Bitwise neural networks for efficient single-channel source separation. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'18).* 701–705.
[20] D. Kleyko, E. P. Frady, and E. Osipov. 2017. Integer echo state networks: Hyperdimensional reservoir computing. *CoRR* abs/1706.00280 (2017).
[21] D. Kleyko, M. Kheffache, E. P. Frady, U. Wiklund, and E. Osipov. 2020. Density encoding enables resource-efficient randomly connected neural networks. *IEEE Trans. Neural Netw. Learn. Syst.* (Early Access). DOI: 10.1109/TNNLS.2020.3015971
[22] S. Ullah, S. Rehman, B. S. Prabakaran, F. Kriebel, M. A. Hanif, M. Shafique, and A. Kumar. 2018. Area-optimized low-latency approximate multipliers for FPGA-based hardware accelerators. In *Design Automation Conference.* 1–6.
[23] 2013. Partial Reconfiguration User Guide. Xilinx. Retrieved on April 26, 2013 from https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug702.pdf, Last accessed on 2020-05-13.
[24] M. Gao and G. Qu. 2020. Estimate and recompute: A novel paradigm for approximate computing on data flow graphs. *IEEE Trans. Comput.-aided Des. Integ. Circ. Syst.* 39, 2 (2020), 335–345.
[25] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *ACM SIGSOFT Symposium.* ACM, 124–134.
[26] I. J. Chang, D. Mohapatra, and K. Roy. 2011. A priority-based 6T/8T hybrid SRAM architecture for aggressive voltage scaling in video applications. *IEEE Trans. Circ. Syst. Vid. Technol.* 21, 2 (2011), 101–112.

[27] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy. 2013. Low-power digital signal processing using approximate adders. *IEEE Trans. Comput.-aided Des. Integ. Circ. Syst.* 32, 1 (2013), 124–137.

[28] S. Mittal. 2016. A survey of techniques for approximate computing. *Comput. Surv.* 48, 4 (2016).

[29] Q. Xu, T. Mytkowicz, and N. S. Kim. 2016. Approximate computing: A survey. *IEEE Des. Test* 33, 1 (2016), 8–22.

[30] A. Agrawal, J. Choi, K. Gopalakrishnan, S. Gupta, R. Nair, J. Oh, D. A. Prener, S. Shukla, V. Srinivasan, and Z. Sura. 2016. Approximate computing: Challenges and opportunities. In *International Conference on Rebooting Computing*. 1–8.

[31] M. Shafique, R. Hafiz, S. Rehman, W. El-Harouni, and J. Henkel. 2016. Invited: Cross-layer approximate computing: From logic to architectures. In *Design Automation Conference*. 1–6.

[32] W. Baek and T. Chilimbi. 2010. Green: A framework for supporting energy-conscious programming using controlled approximation. *SIGPLAN Not.* 45, 6 (June 2010), 198–209.

[33] M. Samadi, J. Lee, D. Jamshidi, A. Hormati, and S. Mahlke. 2013. SAGE: Self-tuning approximation for graphics engines. In *International Symposium on Microarchitecture*. 13–24.

[34] B. Grigorian, N. Farahpour, and G. Reinman. 2015. BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing. In *International Symposium on HPC Architecture*. 615–626.

[35] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke. 2015. Rumba: An online quality management system for approximate computing. In *International Symposium on Computer Architecture*. 554–566.

[36] T. Wang, Q. Zhang, N. Kim, and Q. Xu. 2016. On Effective and efficient quality management for approximate computing. In *International Symposium on Low Power Electronics and Design*. 156–161.

[37] X. Chengwen, W. Xiangyu, Y. Wenqi, X. Qiang, J. Naifeng, L. Xiaoyao, and J. Li. 2017. On quality trade-off control for approximate computing using iterative training. In *Design Automation Conference*. 1–6.

[38] A. Raha, H. Jayakumar, and V. Raghunathan. 2016. Input-based dynamic reconfiguration of approximate arithmetic units for video encoding. *IEEE Trans. Very Large Scale Integ. Syst.* 24, 3 (2016), 846–857.

[39] O. Akbari, M. Kamal, A. Afzali-Kusha, and M. Pedram. 2017. Dual-quality 4:2 Compressors for utilizing in dynamic accuracy configurable multipliers. *IEEE Trans. Very Large Scale Integ. Syst.* 25, 4 (2017), 1352–1361.

[40] J. Pirkl, A. Becher, J. Echavarria, J. Teich, and S. Wildermann. 2017. Self-adaptive FPGA-based image processing filters using approximate arithmetics. In *International Workshop on Software and Compilers for Embedded Systems*. ACM, 89–92.

[41] S. Xu and B. C. Schafer. 2017. Approximate reconfigurable hardware accelerator: Adapting the micro-architecture to dynamic workloads. In *International Conference on Computer Design*. IEEE, 113–120.

[42] S. Xu and B. C. Schafer. 2019. Toward self-tunable approximate computing. *IEEE Trans. Very Large Scale Integ. Syst.* 27, 4 (2019), 778–789.

[43] M. Orlandić and K. Svarstad. 2018. An adaptive high-throughput edge detection filtering system using dynamic partial reconfiguration. *J. Real-Time Image Process.* 16, 1 (2018).

[44] B. Krill, A. Ahmad, A. Amira, and H. Rabah. 2010. An efficient FPGA-based dynamic partial reconfiguration design flow and environment for image and signal processing IP cores. *Sig. Process.: Image Commun.* 25, 5 (2010), 377–387.

[45] M. Nguyen, R. Tamburo, S. Narasimhan, and J. C. Hoe. 2019. Quantifying the benefits of dynamic partial reconfiguration for embedded vision applications. In *International Conference on Field Programmable Logic and Applications*. 129–135.

[46] K. Vipin and S. A. Fahmy. 2018. FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *Comput. Surv.* 51, 4 (2018), 72:1–72:39.

[47] 2019. VC707 Evaluation Board for the Virtex-7 FPGA: User Guide. Xilinx. Retrieved on February 20, 2019 from https://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf.

[48] D. Koch. 2012. *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications.* Springer.

[49] 2019. Xilinx Partial Reconfiguration Controller v1.3, LogiCORE IP Product Guide. Xilinx. Retrieved April 4, 2018 from https://www.xilinx.com/support/documentation/ip_documentation/prc/v1_3/pg193-partial-reconfiguration-controller.pdf.

[50] 2020. AXI HWICAP v3.0: LogiCORE IP Product Guide. Xilinx. Retrieved on October 5, 2016 from https://www.xilinx.com/support/documentation/ip_documentation/axi_hwicap/v3_0/pg134-axi-hwicap.pdf.

[51] 2020. Vivado Design Suite User Guide: Dynamic Function eXchange. Xilinx. Retrieved on January 15, 2020 from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug909-vivado-partial-reconfiguration.pdf.

[52] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke. 2016. Quality control for approximate accelerators by error prediction. *IEEE Des. Test* 33, 1 (2016), 43–50.

[53] S. Shalev-Shwartz and S. Ben-David. 2014. *Understanding Machine Learning: From Theory to Algorithms.* Cambridge University Press.

[54] M. Masadeh, O. Hasan, and S. Tahar. 2019. Using machine learning for quality configurable approximate computing. In *Design, Automation & Test in Europe*. 1554–1557.

[55] P. Ashok, J. Křetínský, K. G. Larsen, A. Le Coënt, J. H. Taankvist, and M. Weininger. 2019. SOS: Safe, optimal and small strategies for hybrid markov decision processes. In *Quantitative Evaluation of Systems*. Springer International Publishing, 147–164.

[56] M. Masadeh, O. Hasan, and S. Tahar. 2019. Input-conscious approximate multiply-accumulate (MAC) unit for energy-efficiency. *IEEE Access* 7 (2019), 147129–147142.

[57] D. Mohapatra, V. K. Chippa, A. Raghunathan, and K. Roy. 2011. Design of voltage-scalable meta-functions for approximate computing. In *Design, Automation Test in Europe*. 1–6.

[58] M. Masadeh, O. Hasan, and S. Tahar. 2019. Error analysis of approximate array multipliers. *CoRR* abs/1908.01343 (2019).

[59] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina. 2017. EvoApprox8B: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In *Design, Automation & Test in Europe*. 258–261.

[60] M. Masadeh, O. Hasan, and S. Tahar. 2020. Machine learning-based self-compensating approximate computing. In *IEEE International Systems Conference*. 1–6.

[61] H. A. F. Almurib, T. N. Kumar, and F. Lombardi. 2016. Inexact designs for approximate low power addition by cell replacement. In *Design, Automation and Test in Europe*. 660–665.

[62] S. García, J. Luengo, and F. Herrera. 2015. *Data Preprocessing in Data Mining*. Springer.

[63] M. Masadeh, A. Aoun, O. Hasan, and S. Tahar. 2020. Decision tree-based adaptive approximate accelerators for enhanced quality. In *IEEE International Systems Conference*. 1–5.

[64] W. J. Chan, A. B. Kahng, S. Kang, R. Kumar, and J. Sartori. 2013. Statistical analysis and modeling for error composition in approximate computation circuits. In *International Conference on Computer Design*. 47–53.

[65] L. Breiman, J. Friedman, R. Olshen, and Ch. Stone. 1984. *Classification and Regression Trees*. Chapman and Hall, Wadsworth.

[66] Su lin Pang and Ji zhang Gong. 2009. C5.0 classification algorithm and application on individual credit evaluation of banks. *Systems Engineering - Theory and Practice* 29, 12 (2009), 94–104.

[67] 2019. The R project for statistical computing. R. Foundation for Statistical Computing. Retrieved on 24 April, 2020 from https://www.r-project.org/.

[68] The MathWorks, Inc. (2018). MATLAB and Classification Learner Toolbox Release. Natick, MA. https://www.mathworks.com/help/stats/classificationlearner-app.html.

[69] The MathWorks, Inc. (2018). MATLAB and HDL Coder Toolbox Release. Natick, MA. The MathWorks, Inc. https://www.mathworks.com/help/stats/classificationlearner-app.html.

[70] 2020. Vivado Design Suite User Guide. xilinx. Retrieved on December 17, 2019 from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug973-vivado-release-notes-install-license.pdf.

[71] 2019. Vivado Design Suite User Guide: Partial Reconfiguration. Xilinx. Retrieved on June 12, 2019 from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug947-vivado-partial-reconfiguration-tutorial.pdf.

[72] P. P. Chu. 2008. *FPGA Prototyping by VHDL Examples: Xilinx Spartan -3 Version*. John Wiley & Sons.

[73] M. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang. 2016. Input responsiveness: Using canary inputs to dynamically steer approximation. In *Programming Language Design and Implementation*. ACM, 161–176.

[74] A. Oliva and A. Torralba. 2001. Modeling the shape of the scene: A holistic representation of the spatial envelope. *Int. J. Comput. Vis.* 42, 3 (2001), 145–175.

[75] A. Oliva and A. Torralba. 2020. Modeling the shape of the scene: A holistic representation of the spatial envelope. Retrieved from http://people.csail.mit.edu/torralba/code/spatialenvelope/.

[76] M. Barni. 2006. *Document and Image Compression*. CRC Press.

[77] 2020. 7 Series FPGAs Data Sheet: Overview. Xilinx. Retreived on September 8, 2020 from https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

[78] K. Papadimitriou, A. Dollas, and S. Hauck. 2011. Performance of partial reconfiguration in FPGA systems: A survey and a cost model. *ACM Trans. Reconfig. Technol. Syst.* 4, 4 (2011), 24.

[79] 2020. BBC Sound Effects. Retrieved from http://bbcsfx.acropolis.org.uk/.

[80] Ch. Solomon. 2011. *Fundamentals of Digital Image Processing a Practical Approach with Examples in Matlab*. Wiley-Blackwell.

Received May 2020; revised October 2020; accepted April 2021