LCS-Based Automatic Configuration of Approximate Computing Parameters for FPGA System Designs

Simon Conrady
Arnold & Richter Cine Technik
Munich, Germany
sconrady@arri.de

Manu Manuel
Technical University of Munich
Munich, Germany
manu.manuel@tum.de

Arne Kreddig
SmartRay GmbH
Wolfratshausen, Germany
arne.kreddig@smartray.com

Walter Stechele
Technical University of Munich
Munich, Germany
walter.stechele@tum.de

ABSTRACT

In application domains like data analysis or image processing, ever-increasing performance demands push the capabilities of computational systems to their limits. With technology scaling plateauing out, engineers are forced to rethink their approach to system design. The research field of approximate computing provides a new design paradigm which trades off accuracy against computational resources. In a complex system, multiple approximation methods can be combined to maximize the resulting benefits, but because of error propagation in the system, doing this in a controlled manner is challenging. To solve this problem, we propose to use concepts developed in the field of evolutionary machine learning to optimize approximation parameters, focusing on systems implemented on FPGA hardware. Our approach uses the rules of a learning classifier system to adjust approximation parameters. The resulting effects on both the application quality and resource usage are estimated on-the-fly and fed back to the rules with every fitness update, allowing the system to be carefully tuned to specific design goals. We illustrate the application of the proposed system to a real-world image processing problem and highlight some practical implications. As this is work in progress, we outline remaining open questions and future directions for our research.

CCS CONCEPTS

• **Computing methodologies** → **Rule learning**; *Learning settings*; Image processing; • **Hardware** → **Reconfigurable logic and FPGAs**;

KEYWORDS

Learning Classifier System, Approximate Computing

GECCO '19 Companion, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6748-6/19/07…$15.00
https://doi.org/10.1145/3319619.3326820

ACM Reference Format:
Simon Conrady, Manu Manuel, Arne Kreddig, and Walter Stechele. 2019. LCS-Based Automatic Configuration of Approximate Computing Parameters for FPGA System Designs. In *Genetic and Evolutionary Computation Conference Companion (GECCO '19 Companion), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3319619.3326820

1 INTRODUCTION

Growing resolution and frame rate of modern imaging systems outpace available resources on current embedded platforms for the efficient implementation of many complex algorithms. Deprivation of the computational capability along with power and energy restrictions in such digital image and signal processing systems extend the focus of research into the area of approximate computing. The idea of approximate computing effectively exploits the inherent resilience of applications to in-exactness in their computations and brings out better performance, space, and energy efficiency on hardware systems which trade off the application quality [7]. However, the quality of such applications must always be preserved above a certain acceptable threshold, and it is essential to identify optimal approximation parameters which meet the quality specifications while maximizing the resource benefits.

Countless research has been conducted on various approximation methods for field-programmable gate array (FPGA) devices over the last decades [20]. However, only a few of them are concentrated on methods to optimize parameters with desired output quality in their practical implementation. Besides, the combination of multiple approximations, the interactions between them, and the effects of the error propagation on the output quality also need to be considered during the implementation of approximations in image and signal processing pipelines. Existing approaches are mostly limited to certain specific approximation methods [1, 30], while our approach can accommodate various methods suitable for use on FPGA devices. Moreover, our rule-based concept can be used to dynamically adjust parameters during runtime.

Previous works have shown the applicability of learning classifier systems (LCSs) to the configuration [8] as well as workload distribution [34] for embedded systems. Inspired by the results of their research, we propose an LCS-based approach which optimizes the parameters of multiple approximations on FPGA devices for image processing applications. The core of the LCS is a set of rules

with conditions and actions collectively modeling an intelligent decision maker along with the fitness function [29]. The actions are used to adjust the configuration of approximation parameters and are chosen depending on conditions which reflect the current system state and also based on fitness values generated during the learning phase. A major benefit of this rule-based approach is that it is not restricted to specific kinds of approximation methods and that it can be used to optimize arbitrary combinations of approximations on system level. Furthermore, the best rules trained in the learning phase can be used to dynamically adjust parameters during runtime without incurring high overhead.

This paper is organized as follows. Section 2 summarizes the related work on approximate computing and the application of LCS to FPGA technology, especially for approximate computing on FPGA devices. In Section 3, background information on LCS-based learning as well as an introduction to FPGA basics are provided. The proposed approach of the LCS-based system is introduced in Section 4, and a potential use case with multiple approximations and optimizations on an image processing pipeline are outlined in Section 5. Section 6 explains the expectations we have for the proposed system and discuss remaining open questions and next steps. Finally, Section 7 concludes our work.

2 RELATED WORK

The error resilience of applications has been effectively exploited with approximate computing by various researchers, especially in image processing applications. A survey on approximate computing by Sparsh Mittal [20] lists a variety of approaches in multiple abstraction levels which trade off computational accuracy against benefits in energy efficiency, resource consumption, and speed. The configurable logic block (CLB) based architecture of the FPGA has been adequately utilized for various approximations such as approximate adders [5, 21], multipliers [27, 28], dividers [3], approximation by wire removal [33], and memorization-based approximation [24]. However, judicious selection of parameters and their optimization are indispensable phases in FPGA-based approximations in order to guarantee the quality specifications along with the performance improvements. If the output quality and resource consumption of an application strongly depend on the application state and the current input, it might be desirable to dynamically change approximation parameters during runtime. This requires a mechanism that should introduce as little overhead as possible to the system.

Danek et al. employed the idea of LCS to FPGA technology mapping problems [8]. Their work introduced a rule-based eXtended classifier system (XCS) adaptive mapper in order to achieve good area and performance results on heterogeneous FPGAs. The mapper is trained on a benchmark circuit and evolves a set of generic mapping rules during the learning phase with minimal number of CLBs and critical signal path delays. Moreover, a reward is generated for each action which reflects the decrease in the number of CLBs and critical path delay along with the utilization of generated CLBs. The adaptive nature of the XCS mapper ensures that the final rules take the global characteristics of the FPGA into account. This work shows that an LCS can be effectively applied to optimize the configuration of FPGA systems. However, it does not consider any approximation methods. In the context of performance and power optimization of System-on-Chips (SoCs), Zeppenfeld et al. introduced the learning classifier table (LCT) [35], a simplified XCS-based reinforcement learning technique, and used the concept for dynamic parameterization to optimize task distribution and workload management in multi-core systems [34]. The LCT monitors the current workload of the cores during runtime, dynamically scales core frequencies, and migrates tasks between cores to achieve optimal system utilization. Their results demonstrate the applicability of an LCS-based approach to the dynamic configuration of embedded systems with low overhead.

Vasicek et al. introduced a two-stage approximation and optimization concept which uses cartesian genetic programming (CGP) on gate level [30]. The objective is to reduce the number of gates in the circuit implemented on the FPGA with a tolerable error rate. To achieve this, the CGP is applied once to the exact circuit to obtain an optimized error-free design first and then again after introducing a certain level of error, resulting in an area-reduced approximate design. This approach directly approximates circuits on gate level and is therefore not applicable to higher level approximation methods. Further, their approach completely eliminates the possibility of dynamic adaption. Akhlaghi et al. proposed an approach that uses gradient descent to introduce multiple approximations in a data flow graph [1]. However, their system can only handle approximate adders and main memory. It is also not suited for dynamic adaption. Regarding the challenge of dynamically adapting approximation parameters, Laurenzano et al. have presented a framework that analyses the current input and searches for the best approximation methods and parameters to adequately process this input [17]. While this approach enables the system to fine tune the approximations to every input, it incurs high runtime overhead and is limited to approximation methods in software.

3 BACKGROUND INFORMATION

This section provides background information on LCSs and presents the major LCS styles. Furthermore, an introduction to the basic concepts of FPGA devices and the implementation of image processing pipelines on FPGA is given.

3.1 LCS

LCS is a methodology in evolutionary machine learning which incorporates rule-based learning. John Holland introduced the concept of LCS in 1976 [10] based on his idea of the genetic algorithm (GA) [11]. The basic LCS involves multiple interacting components and relies on simple rules [29]. These rules are typically combinations of conditions and associated actions which are collectively called a population of classifiers. Generally, rules are defined as *"IF condition, THEN action"*. The driving mechanism of LCSs includes a discovery component and a learning component where the learning component iteratively guides the discovery component to populate a better set of finite rules for a particular problem.

In a general LCS, the environment serves as input to a system which communicates with the classifier population through so-called detectors to identify the matching rules from the population and to form a match set. Appropriate actions are selected from the match set either by random selection or deterministically using a prediction mechanism. Then, effectors perform the selected

actions which in turn change the environment. Different LCSs apply different discovery components such as the GA or the covering mechanism (CM) to the entire population of classifiers, to the match set, or to the selected action set to populate new rules. A reinforcement learning component is responsible for the credit assignment, which is used to update the fitness values that describe how well the rules apply to a problem.

Over the years, the core idea of LCS has been updated with various modifications leading to multiple versions of the LCS [29]. Two major streams of the LCS are the Michigan-style [12] and the Pittsburgh-style [25]. In a Michigan-style LCS, the GA operates at the level of individual rules and the population represents a single solution to the problem. Moreover, each rule in the population has a fixed length. In comparison, a Pittsburgh-style LCS has variable length rule sets which compete with each other. Each of these rule sets is a potential problem solution learned iteratively from a set of problem instances. Hence, this approach could not be adopted to online systems and also needs higher computational effort. Wilson proposed the XCS which is one of the most prominent Michigan-style LCS [31]. Compared to earlier approaches, the XCS separates the credit assignment component from the GA based on accuracy. Also, the rule discovery relies on the selected set rather than the entire population to evolve accurate and general classifiers. The idea of LCS has been adopted in numerous applications such as autonomous robotics [26], data mining [13, 32], traffic signal control [6] and SoC performance optimization [8, 35].

3.2 FPGA

A field-programmable gate array (FPGA) can be used to implement custom logic in hardware. Therefore, FPGAs are commonly used to prototype Integrated Circuits (ICs) or to build small-series products for which the non-recurring engineering costs of manufacturing custom ICs are not economical. Furthermore, in contrast to custom ICs, FPGAs can quickly be reconfigured endless times. As shown in Figure 1, FPGAs consist of three major components: I/O blocks, configurable logic blocks (CLBs), and the interconnect.

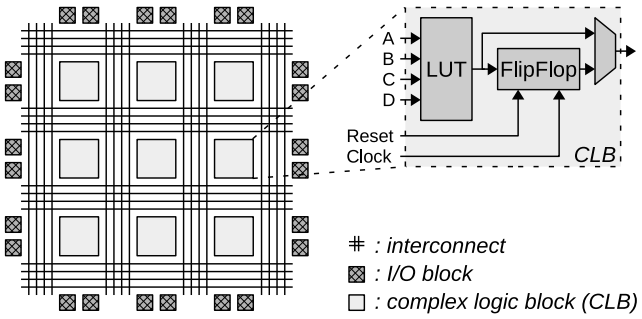

Figure 1: Basic FPGA Structure

I/O blocks are used to connect the FPGA to the outside world. These blocks implement clock and reset connections for the FPGA, as well as data interfaces such as PCIe, SATA, or other data interfaces. FPGAs even support the implementation of full custom interfaces. The actual logic is implemented in the configurable logic blocks (CLBs). CLBs mainly consist of look-up tables (LUTs) holding combinatorial logic and flip-flops to synchronize the logic with the clock. Finally, FPGAs also have a configurable interconnect to connect the CLBs and the I/O blocks with each other. Consequently, the configuration of the CLBs and the interconnect define the logic implemented in the FPGA.

After introducing the basic structure of an FPGA, it is now possible to combine several CLBs to implement more complex logic. This logic is implemented as register-transfer level (RTL) logic, which is depicted exemplarily in Figure 2. In this RTL logic, the data passes through clocked registers and the combinatorial logic between the registers. On every rising clock, the register stores the data available on the input of the register. The stored data then becomes available at the output of the register. The actual calculations on the data are being done in the combinatorial logic between the registers, while the registers just store intermediate results. It is also possible to build complex RTL logic which feeds data back into previous stages. However, to keep the examples simple, more complex logic will not be discussed further here. More information can be found in [16].

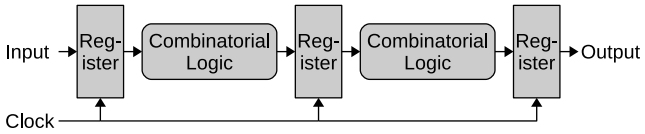

Figure 2: Example of RTL logic

This process can now also be pipelined, so that new input data can be provided to the processing pipeline in every clock cycle. The data will then ripple through the processing pipeline. As no function unit is used twice, there is no need to wait for the previous data set to be processed before feeding new data into the pipeline.

Furthermore, in comparison to traditional CPUs where data always has to be written back into a central register bank or even to the RAM, FPGAs can stream the data through the pipeline without affecting any additional FPGA function units outside the processing pipeline. Hence, FPGAs can achieve a very high throughput.

As current FPGAs integrate between 2,000 to 3,000,000 CLBs, they can integrate very large custom logic. This enables the implementation of complex processing pipelines, such as image processing or 3D data processing pipelines. Further information about the implementation of image processing pipelines in FPGAs can be found in [4].

However, as image processing pipelines are extremely complex, they require lots of FPGA resources. In order to reduce the resource consumption, approximate computing can be applied. There are numerous approximate computing methods available for FPGA. For example, the bitwidth of intermediate results can be changed to optimize the FPGA design. This method, called bitwidth scaling, can be applied very well to FPGA applications, because, unlike CPU architectures, FPGAs are not constrained to a fixed bitwidth. Moreover, these approximation methods have parameters which make the approximation method configurable. In case of bitwidth scaling, a parameter would be the bitwidth which is used for the various signals.

4 PROPOSED APPROACH

We propose a novel approach to optimize the parameters of an approximated FPGA design using an LCS which is based on supervised incremental learning. This section explains the new approach and its components. An overview of the proposed approach is shown in Figure 3.

The system is composed of different components: the approximated application and its parameters which are implemented in FPGA, the LCT which holds all relevant data, and both the Condition and Fitness Function which work on the LCT data to train the system. These components are explained in detail in the following.

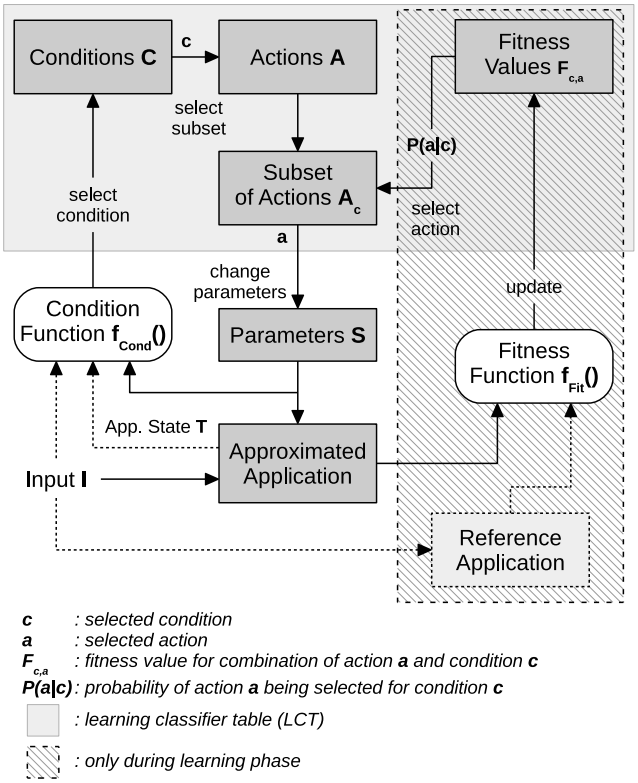

Figure 3: System Overview

The main component of our approach is the approximated FPGA design, called **Approximated Application** in Figure 3. Within the application, multiple approximation methods might be implemented for different tasks. To adjust the approximation, these methods have runtime configurable settings or parameters S. These parameters might control different aspects of the approximation, such as turning individual approximation methods on and off or scaling the precision of intermediate steps in the pipeline [9]. In order to control and tune these parameters, multiple predefined actions A are used. For every action, the LCT stores a fitness value F which is updated after each iteration by the fitness function $f_{Fit}()$. Dependent on this fitness value F, the next action a is chosen accordingly in the next iteration.

As the application runs on an FPGA, our approach can run many iterations per second. Hence, it enables a fast optimization and the exploration of very large design spaces.

The approximation parameters S, which are set by the actions, represent the current state of the approximated cores. Hence, they are also used to select a condition c from a set of possible conditions C. Optionally, the selected condition can also be influenced by the input data I or by the application state T. This application state consists of information offered by the application to aid the learning system in selecting appropriate actions (e.g. average intensity over the last hundred frames in an image processing application). The selected condition c is hence derived as:

$$(S, T, I) \mapsto c \qquad , c \in C. \tag{1}$$

The actual selection of a condition is then done by the condition function $f_{Cond}()$:

$$c = f_{Cond}(S, T, I) \qquad , c \in C. \tag{2}$$

For a set of conditions

$$C = \{c_1, c_2, .., c_n\}, \tag{3}$$

$f_{Cond}()$ contains an entry for every possible condition using S, T, and I to select a condition:

$$f_{Cond}(S, T, I) = \begin{cases} c_1 & \text{, for predefined condition 1} \\ c_2 & \text{, for predefined condition 2} \\ \dots & \dots \\ c_n & \text{, for predefined condition n.} \end{cases} \tag{4}$$

However, the predefined conditions must be constructed in a way so that two predefined conditions can never be true at the same time. Otherwise, $f_{Cond}()$ would not be able to select c reproducibly and reliably.

Then, the selected condition is used to select the subset A_c of valid actions from a set of all possible actions A:

$$\forall c \in C(\exists A_c \subseteq A). \tag{5}$$

To prevent starvation of any action while still ensuring a quick convergence towards an optimal parameter set, actions are chosen randomly with a certain probability $P(a|c)$. This probability depends on the fitness value $F_{c,a}$, which is assigned to every valid combination of conditions and actions:

$$\forall c \in C, a \in A_c(\exists_{=1} F_{c,a} \mapsto P(a|c)). \tag{6}$$

The fitness values together with the conditions and the actions form the LCT, as depicted in Figure 3 and shown exemplarily in Table 1.

As one action is always chosen for any given condition, the sum over the probabilities for all valid actions is always 100% for all possible conditions:

$$\sum_{a}^{A_c} P(a|c) = 1 \qquad , \forall c \in C. \tag{7}$$

Consequently, the probability for an action to get chosen is calculated as:

$$P(a|c) = \frac{F_{c,a}}{\sum_{a_k}^{A_c} F_{c,a_k}} \qquad , \sum_{a_k}^{A_c} F_{c,a_k} \geq F_{c,a} > 0. \tag{8}$$

Therefore, no fitness value $F_{c,a}$ is allowed to be <= 0. Otherwise, such a fitness value would always result in a zero probability of the assigned action being chosen. Hence, the action would starve.

To calculate the fitness value $F_{c,a}$, the benefits and drawbacks of applying the action in the last iteration of the learning system are taken into account. This is done by the fitness function $f_{Fit}()$, which decides whether it was good or bad to apply the action a under the given condition c. If it was good to apply the action, the fitness value will be increased, otherwise, it will be decreased. Benefits of applying the action might be less power consumption, less area usage, lower latency, or any other application specific benefit. Drawbacks on the other hand represent the decrease in data quality at the output of the application. These benefits and drawbacks span the design space we are exploring with our approach. To calculate the new fitness value $F^{new}_{c,a}$, we now take the old fitness value $F^{old}_{c,a}$, the set of all benefits $B_{c,a}$, and the set of all drawbacks $D_{c,a}$ into account:

$$F^{new}_{c,a} = f_{Fit}(F^{old}_{c,a}, B_{c,a}, D_{c,a}). \tag{9}$$

To evaluate the output quality of the approximated application, a reference design can be implemented in parallel to the approximated design. This reference design, however, is optional and the quality of the output data could also be quantified with a model-based approach.

Furthermore, some benefits such as power consumption or area usage might not or even cannot be measured at runtime. Instead, a model of the benefit can be used to calculate the fitness function. This model is part of the fitness function $f_{Fit}()$.

An abstract example of a learning classifier table is shown in Table 1. This example is based on a set of conditions $C = \{c_1 ... c_n\}$ and actions $A = \{a_1 ... a_7\}$. For every condition c, the second column lists the subset A_c of all actions which are valid for the selected condition. The third column then holds the calculated fitness value accordingly.

Table 1: Example LCS Table

Condition	Action	Fitness
c_1	a_1	F_{11}
	a_2	F_{12}
	a_5	F_{15}
	a_7	F_{17}
c_2	a_1	F_{21}
	a_3	F_{23}
	a_4	F_{24}
	a_5	F_{25}
	a_6	F_{26}
...	...	...
c_n	$\{a_m : A_{c_n}\}$, $A_{c_n} \subseteq \{a_1 ... a_7\}$	$\{F_{n,m}\}$

Using this approach, we can now optimize the parameters of an approximated FPGA design. Our approach can be used to optimize the parameters either statically or dynamically.

When using static optimization, the approximation parameters are learned in the first phase, called learning phase. Afterwards, in the production phase, the parameter set is being fixed, so the system always uses these optimized parameters.

When using dynamic optimization, the approximation parameters are optimized in the learning phase as explained before. In the production phase, however, the fitness value $F_{c,a}$ is not updated anymore. Hence, the fitness function $f_{Fit}()$ is not used in the production phase. Actions are now chosen directly by the highest learned fitness value $F_{c,a}$ from each action set A_c instead of choosing randomly. Consequently, all actions with lower fitness values can now be omitted, as they won't be chosen anyway. The set A_c now only contains one action:

$$|A_c| = 1 \qquad , \forall c \in C. \tag{10}$$

Hence, this action has a probability $P(a|c) = 1$ of being chosen. As a result, the fitness values can now be completely omitted in the production phase.

When using dynamic optimization, the influence of the input data I and the application state T might be especially interesting, as this application now allows adjusting the approximation parameters depending on the input data even in the production phase.

The proposed approach is not directly equivalent to any of the existing LCSs. However, it inherits basic concepts from the Michigan-style LCS by using a finite rule set population that represents the problem solution [12]. Like Zeppenfeld et al. [35], we derive probabilities from the fitness values for the prediction mechanism. The condition function plays the role of the detectors which populates the match set rules with the relevant actions. In contrast to general LCSs, all conditions are mutually exclusive in our approach, but multiple different rules share the same condition. Therefore, the resulting match set can not contain duplicate actions. Our initial concept uses a deterministic set of rules, but we are evaluating the possibility of integrating the genetic mechanism as well. However, we are using reinforcement-based credit assignment to iteratively update the fitness values of selected rules with benefits and drawbacks to end up with an optimal rule set which can be used for parameter adjustments at runtime.

5 DISPLAY RENDERING SCENARIO

As a suitable use case for our proposed optimization system we consider a display rendering pipeline as employed by typical digital stills or motion picture cameras [2]. In this pipeline, a series of tasks is performed to adapt an input image for display on a monitor, adapting the image to the dynamic range capabilities, the color space and the electro-optical transfer function (EOTF) of the monitor. The pipeline contains highly non-linear functions which cannot easily be computed in camera hardware and are therefore implemented as look-up tables (LUTs). To reduce resource demand, we apply approximate computing techniques and use the proposed approach to optimize the approximation parameters.

5.1 The Processing Pipeline

The rendering pipeline is depicted in Figure 4. It consists of three steps. The input to the system is image data in a scene-referred encoding that relates to real-world luminances and colors captured

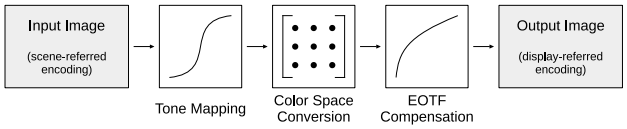

Figure 4: Processing pipeline for display rendering scenario

by the camera. Each pixel consists of a triplet of red, green and blue luminances, denoted by $[R_{in}G_{in}B_{in}]^\top$.

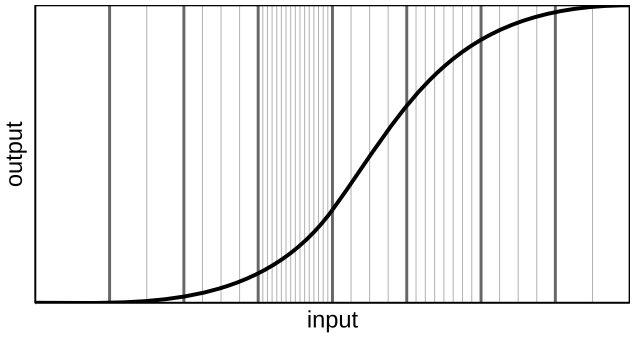

Figure 5: Tone-mapping curve ($h = 9$, $k = 0.6$, $I_a = 0.4$ and $\mathrm{enc}^{-1}(x) = 2^x$) and exemplary hierarchical segmentation (the thick and thin vertical lines denote section and interval boundaries, respectively)

First, a tone-mapping operator is used to map the input luminance to display luminance. The aim of this step is to create a visually natural reproduction of the captured scene on a display with limited dynamic range. An overview of the research on tone-mapping operators can be found in [23]. For this example, we use a global sigmoidal operator that is based on a model of photoreceptor behavior [22]. The tone-mapped pixel values C_{tm} are calculated as

$$C_{tm} = \frac{\mathrm{enc}^{-1}(C_{in})}{\mathrm{enc}^{-1}(C_{in}) + (hI_a)^k} s + o, \tag{11}$$

where $C \in \{R, G, B\}$ represents the luminance in any color channel, and I_a is the adaption level in the model, which we set to the level to which the input scene-encoding maps neutral grey. Both h and k can be seen as user parameters used to control the overall luminance and the contrast of the output. Because the employed tone-mapping function is defined for linear intensities, we have to linearize the input values by using the inverse of the logarithmic intensity encoding (enc^{-1}). The variables s and o are used to scale and shift the output to the desired range.

After the tone-mapping step, the image is transformed to the color space of the display by multiplication of the tone-mapped color channels with a conversion matrix:

$$\begin{bmatrix} R_{col} \\ G_{col} \\ B_{col} \end{bmatrix} = \underbrace{\begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix}}_{\mathbf{M}} \begin{bmatrix} R_{tm} \\ G_{tm} \\ B_{tm} \end{bmatrix}, \tag{12}$$

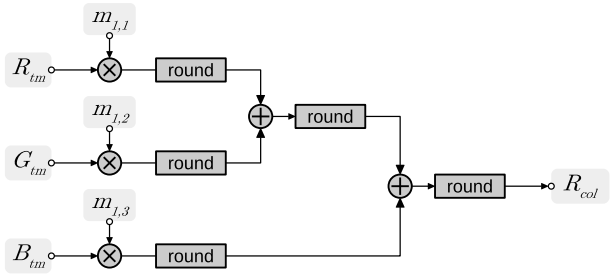

Figure 6: Signal flow of the matrix transformation (only one output channel shown)

where $[R_{col}G_{col}B_{col}]^\top$ is the resulting triplet in the target color space. The matrix entries $m_{i,j}$ depend on the primaries of the input encoding color space as well as those of the display color space. For this example, we use the conversion from Alexa Wide Gamut [2] to the Rec.709 color space [15], with the matrix

$$\mathbf{M}_{709} = \begin{bmatrix} 1.6175 & -0.5373 & -0.0802 \\ -0.0706 & 1.3346 & -0.2640 \\ -0.0211 & -0.2270 & 1.2481 \end{bmatrix}. \tag{13}$$

The last step in the pipeline compensates for the displays' EOTF. Corresponding transfer functions are defined in various standards and recommendations. In IEC 61966-2-1 Amendment 1 [14], the transfer function is defined as

$$C_{out} = \begin{cases} 12.92\, C_{col} & , \text{for } C_{col} < 0.0031308 \\ 1.055\, C_{col}^{1/2.4} - 0.055 & , \text{for } C_{col} \geq 0.0031308, \end{cases} \tag{14}$$

where C represents any of $\{R, G, B\}$. The output values C_{out} are now adapted to the characteristics of a specific display type.

5.2 Hardware Implementation and Approximation Techniques

In this example, both the input and output bitwidth of the pipeline are set to 16 bit. In the reference implementation, the tone-mapping and EOTF compensation are realized as full LUTs, i.e. for all possible input values the respective output is stored in memory (with 16 bit precision), taking up 131072 bytes of memory. The matrix transformation is implemented in fixed-point arithmetic. The signal flow for the calculation can be seen in Figure 6. All matrix entries are stored as 24 bit values with 2 integer bits (including the sign) and 22 fractional bits, and the output is rounded to 16 bit while no rounding takes place in intermediate steps.

For the approximated version of the pipeline, we replace the full LUTs with hierarchically segmented sparse LUTs as proposed by Lee [18]. We use a two-level segmentation scheme where both the outer segments (further denoted as sections) and the inner segments (further denoted as intervals) are uniformly distributed. The hierarchical scheme allows for an optimized distribution of grid points depending on the represented function. An exemplary segmentation of the tone-mapping LUT can be seen in Figure 5. The function output is computed by using the most significant part of the input to address the LUT data and subsequent interpolation

using the remaining bits. Different interpolation methods can be used, e.g. nearest neighbor or linear interpolation. These modules can be further approximated by reducing the bitwidth of the stored data.

In the matrix transformation step, we introduce multiple approximation methods: reducing the precision of stored matrix values, rounding or cutting bits in intermediate results and replacing multipliers or adders with approximated units. These units often expose further approximation parameters (e.g. the position where a carry chain is cut in an approximate adder [19]).

5.3 Parameter Optimization Using the Proposed System

Introducing approximation techniques like the ones mentioned above exposes numerous design parameters. While the approximations yield benefits in terms of resource usage, they also introduce error to the signal which needs to be controlled to ensure an acceptable quality of service. This becomes difficult for complex systems as multiple errors introduced to different signals may interact with each other and propagate through the processing pipeline. It is therefore not sufficient to optimize individual signals or modules on their own. To account for interactions of the used approximations, we use the proposed LCS for a global optimization across the complete pipeline by observing the errors at the final output.

We use the actions of the LCS to gradually change the system configuration, adjusting one parameter at a time, until the system converges towards an optimal state as guided by the fitness values. Table 2 lists all possible actions for the approximation techniques used in this example. Some of these actions are not necessarily always available, depending on the current parameter configuration S. For example, in a sparse LUT, the number of intervals in a specific section can only be adjusted when that section exists in the current configuration. To prevent the system from taking incorrect actions, the LCT only lists condition-action tuples (c, a) for which the action a is possible when condition c is true.

Table 2: Approximation parameters and corresponding LCS actions for the rendering pipeline

Module	Parameters	Actions
Sparse Luts	No. of sections	Inc/Dec
	No. of intervals in section x	Inc/Dec
	Precision of stored values	Inc/Dec
	Interpolation method	Change
Color Matrix	No. of bits for matrix entry $m_{i,j}$	Inc/Dec
	No. of bits for intermediate result x	Inc/Dec
	Rounding method at position x	Change
	Implementation of arithmetic unit x	Change
	Approximation parameter of arithmetic unit x	Inc/Dec/ Change

In this example we use solely the parameter configuration S to select conditions (so that $c = f_{Cond}(S)$, compare Equation 2). We use the condition function to cluster the parameter space into several subspaces by grouping ranges of parameters into the same condition. This allows us to keep the number of conditions at a reasonable level while still being able to record different fitness values depending on the parameter configuration. When, as for this example, the conditions depend exclusively on the parameter configuration S, the optimization is strongly tied to the test set I^{test} for which the quality is estimated. Therefore, we use a generic test set consisting of uniformly distributed colors to ensure neutral conditions and to catch corner cases. Alternatively, real images with specific content (e.g. landscapes or portraits) could be used as test set to obtain a content-specific parameter configuration. Because the complete processing of the test set in both the reference and approximated implementation takes place on an FPGA, the system can run large numbers of iterations even with big test sets within reasonable time.

With each execution of an action, we update the corresponding fitness value. For the example system, we consider three performance indicators: the decrease in both memory consumption and computational area as benefits and the loss in image quality as drawback.

The memory consumption benefit B^{mem} is calculated as the difference in memory consumption for the LUTs between the reference and approximated design:

$$B^{mem} = \sum_{l \in L} \left(\text{bits}^{ref}(l) - \text{bits}^{ax}(l) \right), \tag{15}$$

where bits(l) denotes the number of bits used to store the LUT l and the superscripts ref and ax refer to the reference and approximated implementation, respectively. L is the set of all LUTs in the design. Regarding the computational area, we divide the system into functional units u, and calculate the corresponding benefit B^{comp} in a similar manner:

$$B^{comp} = \sum_{u \in U} \left(\text{CLB}^{ref}(u) - \text{CLB}^{ax}(u) \right), \tag{16}$$

where CLB(u) denotes the number of CLBs used to implement functional unit u and U is the set of all units.

The loss in quality is modeled with metrics derived from the error between the reference output C^{ref}_{out} and the approximated output C^{ax}_{out}. For any single pixel i in a test set I^{test}, the error in color channel $C \in \{R, G, B\}$ is calculated as

$$\Delta C(i) = C^{ref}_{out}(i) - C^{ax}_{out}(i). \tag{17}$$

For our quality model, we use the following three derived metrics: the mean squared error

$$E^{mse} = \frac{1}{3|I^{test}|} \sum_{C \in \{R,G,B\}} \sum_{i \in I^{test}} \Delta C(i)^2 \tag{18}$$

as a general indication of quality loss, the maximum absolute error

$$E^{wc} = \max_{C \in \{R,G,B\}} \max_{i \in I^{test}} \text{abs}\left(\Delta C(i)\right), \tag{19}$$

which represents the worst case error, and the absolute bias

$$E^{bias} = \text{abs}\left(\frac{1}{3|I^{test}|} \sum_{C \in \{R,G,B\}} \sum_{i \in I^{test}} \Delta C(i) \right), \tag{20}$$

which indicates a shift in overall luminance.

The overall quality drawback estimate is a weighted sum of these errors:

$$D^{err} = w_{mse}E^{mse} + w_{wc}E^{wc} + w_{bias}E^{bias}, \tag{21}$$

where the weights w give the system designer a way to control the importance of each error type.

For the update of the fitness value associated with the condition-action tuple (c, a), another weighting scheme is used to combine the benefits and drawbacks into a single value representing the reward of the last application of (c, a):

$$R_{c,a} = w_{mem}B^{mem}_{c,a} + w_{comp}B^{comp}_{c,a} - w_{err}D^{err}_{c,a}, \tag{22}$$

where the weights w control the scale and importance of the different benefits and the quality drawbacks according to the overall design goals. When the benefits outweigh the drawbacks, we want the fitness value to increase and vice versa. To achieve this, we use a simple update function for Equation 9:

$$F^{new}_{c,a} = f_{Fit}(F^{old}_{c,a}, B_{c,a}, D_{c,a}) = F^{old}_{c,a}\alpha^{R_{c,a}}, \tag{23}$$

where α controls the rate at which the fitness values are adapted. In the beginning of the optimization, we have no prior knowledge about the performance of any action. Therefore, we initialize all fitness value with the same start value:

$$F^{init}_{c,a} = 100 \qquad ,\forall c \in C, a \in A_c. \tag{24}$$

When applying Equation 8 to assign choice probabilities, this automatically ensures that every action a given the current condition c is chosen with equal probability at the start.

6 FUTURE WORK

The main body of this work presents the concept for LCS-based optimization of FPGA systems that incorporate approximate computing methods, and it is work in progress. As next steps, we plan to implement the system and evaluate its utility on use cases like the one presented in Section 5. Furthermore, we want to address several remaining open questions which are outlined in the following.

When using the proposed system for static optimization of an approximated FPGA system, we expect the learned parameters to balance the different approximations for good overall system performance. With generic test data for the training, the quality is controlled for any input that the application encounters in its use, including corner cases. The model for the fitness update gives the system designer a way to adjust the optimization according to specific design goals. However, the choice of weights for the benefits and drawbacks is strongly application-specific and might not be immediately obvious. Experiments with different applications and approximation methods could provide insights and guidance for choosing these weights. Furthermore, although we expect the proposed fitness update function to yield useful results for the exemplary application, there might well be room for improvements because of its simple nature. Such improvements can draw from ongoing research on credit assignment in LCS [29].

The case study presented in Section 5 does not consider a dynamic optimization of the application. However, the method proposed in this paper is designed to also support such a use case. To do so, the condition function $f_{cond}()$ must also include a dependency on the current input I and/or the application state T. In the rendering pipeline, a goal could be that the system dynamically adapts to the average intensity of the images. The application state T could store a history of average intensity values from past frames, which could be used to eliminate outliers. By including the history stored in T and the current value drawn from I, the LCS could, for example, shift more grid points in the sparse LUTs towards the average intensity. This necessitates that in the learning phase, the same condition function is used to find the best rules depending on the input and application state. At runtime, the system uses the pre-trained rules to change approximation parameters on-the-fly. This means that only approximations suitable for dynamic adaption can be used, and the resource overhead introduced by the dynamic adaption mechanism must stay below the anticipated benefits.

In general, the condition function unambiguously maps the parameter configuration, the current input, and the application state to a distinct condition. The boundaries used for this mapping currently have to be provided by the system designer, but the GA could be used to learn optimal boundaries. Also, it must be carefully considered that hard boundaries resulting from the division of the respective input spaces could lead to an oscillating system. A possible solution to this problem could be using a hysteresis in the condition function to prevent oscillation.

7 CONCLUSION

In this paper, we present initial ideas about a new framework for the optimization of approximate computing parameters for FPGA system designs based on a novel approach using an LCS. Our approach does not only aim at the optimization of approximation parameters in small subsets or functions of FPGA designs but also enables the optimization of large parts of the FPGA design on system level, including error propagation through the system. Furthermore, the presented approach can optimize the approximation parameters either statically or dynamically. Hence, it also allows the design of parameter sets which can adapt to the application input even after the learning phase has been completed.

Moreover, we illustrate the application of the proposed system to an image processing pipeline. This scenario integrates sparse LUTs, bitwidth scaling, different rounding methods, and approximated arithmetic units. We propose the optimization of the approximation parameters associated with these methods using our framework.

As part of our future work, we plan to do a proof of concept implementation, including a thorough evaluation. Furthermore, we plan to integrate more sophisticated calculations to the fitness function and to investigate how the condition space can be segmented without the system starting to oscillate around the boundaries.

ACKNOWLEDGMENTS

This work is supported by the Bavarian Ministry of Economic Affairs, Regional Development and Energy under Grant No.: IUK574 (https://www.iuk-bayern.de).

REFERENCES

[1] Vahideh Akhlaghi, Sicun Gao, and Rajesh K. Gupta. 2018. LEMAX: Learning-based Energy Consumption Minimization in Approximate Computing with Quality Guarantee. In *Proceedings of the 55th Annual Design Automation Conference (DAC '18)*. ACM, 161:1–161:6. https://doi.org/10.1145/3195970.3196069

[2] Stefano Andriani, Harald Brendel, Tamara Seybold, and Joseph Goldstone. 2013. Beyond the Kodak Image Set: A New Reference Set of Color Image Sequences. In *2013 IEEE International Conference on Image Processing*. IEEE, 2289–2293. https://doi.org/10.1109/ICIP.2013.6738472

[3] Donald G. Bailey. 2006. Space Efficient Division on FPGAs. In *Proceedings of the Thirteenth Electronics New Zealand Conference ENZCon 06*. Electronics New Zealand Inc., 206–211. http://seat.massey.ac.nz/research/centres/SPRG/pdfs/2006_ENZCon_206.pdf

[4] Donald G. Bailey. 2011. *Design for Embedded Image Processing on FPGAs*. John Wiley & Sons (Asia), Singapore. https://doi.org/10.1002/9780470828519

[5] Andreas Becher, Jorge Echavarria, Daniel Ziener, Stefan Wildermann, and Jürgen Teich. 2016. A LUT-Based Approximate Adder. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 27. https://doi.org/10.1109/FCCM.2016.16

[6] Larry Bull, Jeanan Sha'Aban, Andy Tomlinson, J. D. Addison, and Benjamin G. Heydecker. 2004. *Towards Distributed Adaptive Control for Road Traffic Junction Signals using Learning Classifier Systems*. Springer Berlin Heidelberg, 276–299. https://doi.org/10.1007/978-3-540-39925-4_12

[7] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Analysis and Characterization of Inherent Application Resilience for Approximate Computing. In *Proceedings of the 50th Annual Design Automation Conference (DAC '13)*. ACM, 113:1–113:9. https://doi.org/10.1145/2463209.2488873

[8] Martin Danek and Robert E. Smith. 2002. XCS Applied to Mapping FPGA Architectures. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation (GECCO'02)*. Morgan Kaufmann Publishers Inc., 912–919. http://dl.acm.org/citation.cfm?id=2955491.2955655

[9] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE, 449–460. https://doi.org/10.1109/MICRO.2012.48

[10] John H. Holland. 1976. Adaptation. In *Progress in Theoretical Biology*, Robert Rosen and Fred M. Snell (Eds.). Academic Press, 263–293. https://doi.org/10.1016/B978-0-12-543104-0.50012-3

[11] John H. Holland. 1992. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press. https://doi.org/10.7551/mitpress/1090.001.0001

[12] John H. Holland and Judith S. Reitman. 1978. Cognitive Systems Based on Adaptive Algorithms. In *Pattern-Directed Inference Systems*, Donald Arthur Waterman and Frederick Hayes-Roth (Eds.). Academic Press, 313 – 329. https://doi.org/10.1016/B978-0-12-737550-2.50020-8

[13] John H. Holmes and Jennifer A. Sager. 2005. Rule Discovery in Epidemiologic Surveillance Data Using EpiXCS: An Evolutionary Computation Approach. In *Artificial Intelligence in Medicine*, Silvia Miksch, Jim Hunter, and Elpida T. Keravnou (Eds.). Springer Berlin Heidelberg, 444–452. https://doi.org/10.1007/11527770_60

[14] IEC. 2003. *IEC 1966-2-1:1999/AMD1:2003: Amendment 1: Multimedia Systems and Equipment –Colour Measurement and Management –Part 2-1: Colour Management –Default RGB Colour Space - sRGB*. International Electrotechnical Commission. https://webstore.iec.ch/publication/6168

[15] ITU. 2015. *Recommendation ITU-R BT.709-6: Parameter Values for the HDTV Standards for Production and International Programme Exchange*. International Telecommunication Union. https://www.itu.int/rec/R-REC-BT.709

[16] Dirk Koch, Daniel Ziener, and Frank Hannig. 2016. *FPGAs for Software Programmers*. Springer, Cham. https://doi.org/10.1007/978-3-319-26408-0_1

[17] Michael A. Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. 2016. Input Responsiveness: Using Canary Inputs to Dynamically Steer Approximation. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, 161–176. https://doi.org/10.1145/2908080.2908087

[18] Dong-U Lee, Ray C. C. Cheung, Wayne Luk, and John D. Villasenor. 2009. Hierarchical Segmentation for Hardware Function Evaluation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17, 1 (Jan 2009), 103–116. https://doi.org/10.1109/TVLSI.2008.2003165

[19] Hamid R. Mahdiani, Ali Ahmadi, Sied M. Fakhraie, and Caro Lucas. 2010. Bio-Inspired Imprecise Computational Blocks for Efficient VLSI Implementation of Soft-Computing Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers* 57, 4 (Apr 2010), 850–862. https://doi.org/10.1109/TCSI.2009.2027626

[20] Sparsh Mittal. 2016. A Survey of Techniques for Approximate Computing. *ACM Computing Surveys* 48, 4 (Mar 2016), 62:1–62:33. https://doi.org/10.1145/2893356

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

[22] Erik Reinhard and Kate Devlin. 2005. Dynamic Range Reduction Inspired by Photoreceptor Physiology. *IEEE Transactions on Visualization and Computer Graphics* 11, 1 (Jan 2005), 13–24. https://doi.org/10.1109/TVCG.2005.9

[23] Erik Reinhard, Wolfgang Heidrich, Paul Debevec, Sumanta Pattanaik, Greg Ward, and Karol Myszkowski. 2010. *High Dynamic Range Imaging* (first ed.). Morgan Kaufmann. https://www.elsevier.com/books/high-dynamic-range-imaging/reinhard/978-0-12-374914-7

[24] Sharad Sinha and Wei Zhang. 2016. Low-Power FPGA Design Using Memoization-Based Approximate Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 8 (Aug 2016), 2665–2678. https://doi.org/10.1109/TVLSI.2016.2520979

[25] Stephen Frederick Smith. 1980. *A Learning System Based on Genetic Adaptive Algorithms*. Ph.D. Dissertation. https://dl.acm.org/citation.cfm?id=909835 AAI8112638.

[26] Wolfgang Stolzmann and Martin Butz. 2000. Latent Learning and Action Planning in Robots with Anticipatory Classifier Systems. In *Learning Classifier Systems*, Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson (Eds.). Springer Berlin Heidelberg, 301–317. https://doi.org/10.1007/3-540-45027-0_16

[27] Salim Ullah, Sanjeev Sripadraj Murthy, and Akash Kumar. 2018. *SMApproxlib*: library of FPGA-based approximate multipliers. In *Proceedings of the 55th Annual Design Automation Conference (DAC '18)*. ACM, 157:1–157:6. https://doi.org/10.1145/3195970.3196115

[28] Salim Ullah, Semeen Rehman, Bharath Srinivas Prabakaran, Florian Kriebel, Muhammad Abdullah Hanif, Muhammad Shafique, and Akash Kumar. 2018. Area-optimized low-latency approximate multipliers for FPGA-based hardware accelerators. In *Proceedings of the 55th Annual Design Automation Conference (DAC '18)*. ACM, 159:1–159:6. https://doi.org/10.1145/3195970.3195996

[29] Ryan J. Urbanowicz and Jason H. Moore. 2009. Learning Classifier Systems: A Complete Introduction, Review, and Roadmap. *Journal of Artificial Evolution and Applications* 2009 (June 2009), 1–25. https://doi.org/10.1155/2009/736398

[30] Zdenek Vasicek and Lukas Sekanina. 2016. Search-based synthesis of approximate circuits implemented into FPGAs. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–4. https://doi.org/10.1109/FPL.2016.7577305

[31] Stewart W. Wilson. 1995. Classifier Fitness Based on Accuracy. *Evolutionary Computation* 3, 2 (Jun 1995), 149–175. https://doi.org/10.1162/evco.1995.3.2.149

[32] Stewart W. Wilson. 2001. Mining Oblique Data with XCS. In *Advances in Learning Classifier Systems*, Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson (Eds.). Springer Berlin Heidelberg, 158–174. https://doi.org/10.1007/3-540-44640-0_11

[33] Yi Wu, Chuyu Shen, Yi Jia, and Weikang Qian. 2017. Approximate logic synthesis for FPGA by wire removal and local function change. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 163–169. https://doi.org/10.1109/ASPDAC.2017.7858314

[34] Johannes Zeppenfeld, Abdelmajid Bouajila, Walter Stechele, Andreas Bernauer, Oliver Bringmann, Wolfgang Rosenstiel, and Andreas Herkersdorf. 2012. Applying ASoC to Multi-core Applications for Workload Management. In *Autonomic Systems, 1, Volume 1, Organic Computing - A Paradigm Shift for Complex Systems, Part 5*. Springer Basel, Pages 461–472. https://doi.org/10.1007/978-3-0348-0130-0_30

[35] Johannes Zeppenfeld, Abdelmajid Bouajila, Walter Stechele, and Andreas Herkersdorf. 2008. Learning Classifier Tables for Autonomic Systems on Chip. In *GI Jahrestagung*. Gesellschaft für Informatik, 771–778. https://subs.emis.de/LNI/Proceedings/Proceedings134/article2181.html