Approximate Logic Synthesis: A Survey

Ilaria Scarabottolo, *Member, IEEE,* Giovanni Ansaloni, *Member, IEEE,* George A. Constantinides, *Senior Member, IEEE,* Laura Pozzi, *Member, IEEE,* and Sherief Reda, *Senior Member, IEEE*

***Abstract*—Approximate computing is an emerging paradigm that, by relaxing the requirement for full accuracy, offers benefits in terms of design area and power consumption. This paradigm is particularly attractive in applications where the underlying computation has inherent resilience to small errors. Such applications are abundant in many domains, including machine learning, computer vision and signal processing. In circuit design, a major challenge is the capability to synthesize approximate circuits *automatically*, without manually relying on the expertise of designers. In this work, we review methods devised to synthesize approximate circuits given their exact functionality and an approximability threshold. We summarize strategies for evaluating the error that circuit simplification can induce on the output, which guide synthesis techniques in choosing the circuit transformations that lead to the largest benefit for a given amount of induced error. We then review circuit simplification methods that operate at gate or Boolean level, including those that leverage classical Boolean synthesis techniques to realize the approximations. We also summarize strategies that take high-level descriptions such as C or behavioral Verilog and synthesize approximate circuits from these descriptions.**

***Index Terms*—Approximation, Circuit, Logic Synthesis**

I. Introduction

Given an intended functionality, established methodologies for hardware design focus on achieving good trade-offs between performance metrics (*e.g.* latency, throughput) and cost (energy and resource requirements). Hence, higher-performance circuits can only be obtained by increasing their size or power budget, while the relationship between inputs and output values is kept invariant. Approximate Logic Synthesis (ALS) expands the scope of this process by adding a further dimension to the design space of possible solutions: that of the tolerated implementation inaccuracy, as illustrated in Figure 1b.

Approximate hardware components realized with ALS can, at the same time, offer remarkable gains in area and efficiency *and* significant performance increases with respect to their exact counterparts, in exchange for small losses in output quality. ALS is hence the embodiment, at the hardware design level, of Approximate Computing (AC). The AC paradigm investigates the benefit of judicious Quality-of-Result (QoR) degradations in different levels of the hardware/software stack, spanning from software solutions, through the design of digital architectures, to *circuit design*, as illustrated in Figure 1a. This survey covers the approximate circuit design techniques; readers interested in AC methodologies and solutions in a broader context can refer to the recent surveys by Liu *et al.* [29] Han *et al*. [14] and Xu *et al*. [61]. Approximate circuit synthesis is particularly attractive since approximate circuits are employed as basic blocks for realizing application-specific accelerators, which are a highly relevant component of modern Systems-on-Chips [18].

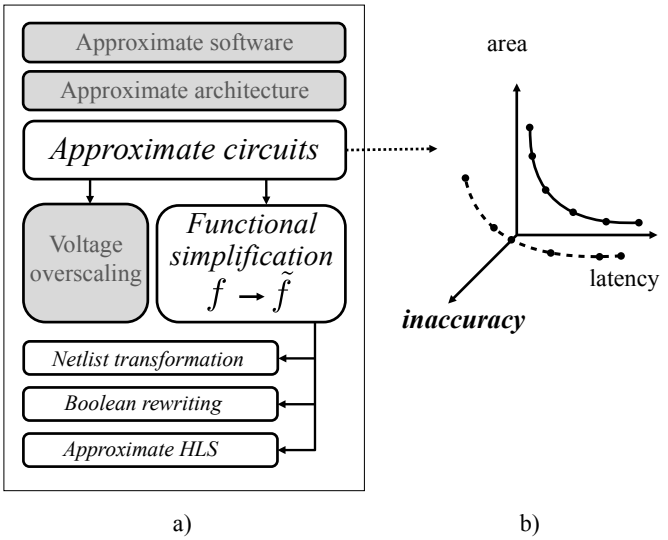


Fig. 1: (a) Approximate Computing (AC) techniques taxonomy, where areas discussed in this paper are highlighted. (b) Inaccuracy represents a new dimension for circuit synthesis.

Approximate circuit synthesis has the ability to automate the process of discovering approximate implementations given an exact circuit description. For example, we provide in Figure 2 an illustration of the ability of approximate synthesis techniques to generate a large number of approximate variants of an 8-point Fast Fourier Transform (FFT) circuit, where each point reports the results of one approximate design. For Figure 2, we use the open-source tool ABACUS [38] together with the FreePDK 45 nm library [52]. We use four test waveforms to evaluate the amplitude spectrum of both the original circuit and the approximate variants, and we report the mean square error (MSE) relative to the original spectrum in percentage. The results in the Figure show the potential of automated approximate logic synthesis: we can achieve large reductions of area savings with negligible reduction in QoR. For instance, we can save 22% of the design area at the expense of 0.12% reduction in accuracy. The FFT circuit is part of the Benchmarks for Approximate Circuit Synthesis (BACS) that we release in conjunction with this paper.[1]

Ilaria Scarabottolo, Giovanni Ansaloni and Laura Pozzi are with the Faculty of Informatics, USI Lugano, Switzerland. George Constantinides is with Imperial College London, UK. Sherief Reda is with Brown University, USA. Contact Author: sherief_reda@brown.edu.

[1]BACS benchmark set: https://github.com/scale-lab/BACS.

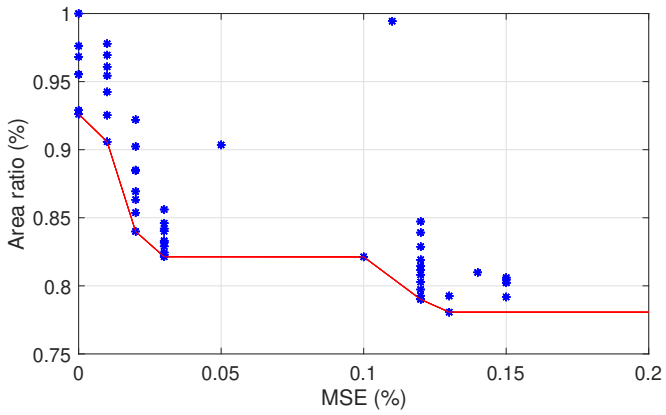


Fig. 2: Approximate design variants of an 8-point FFT circuit generated using the ABACUS tool. Each point represents the error and design area of an approximate circuit compared to the original design. MSE stands for Mean Square Error.

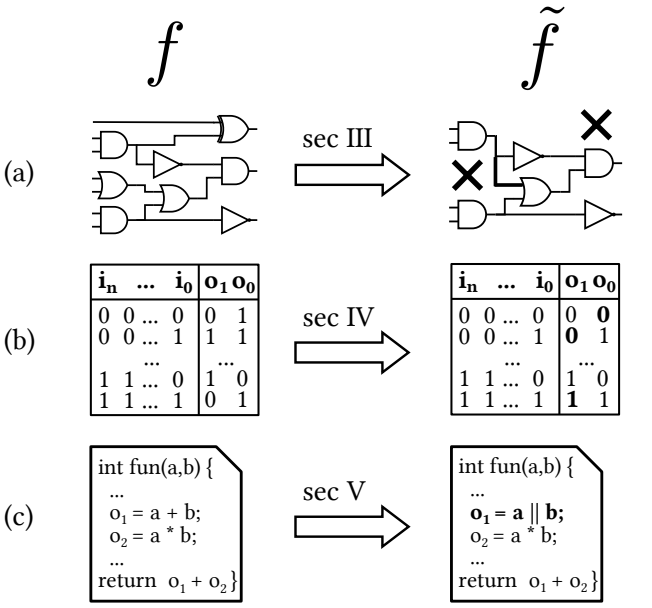


Fig. 3: Possible functional simplification approaches are illustrated: a generic Boolean function f is transformed in $\tilde{f}$, either acting on a synthesised netlist, for instance, at gate-level, or on its truth table. Finally, the circuit at behavioural level can be simplified by approximate high-level synthesis.

We survey synthesis strategies for automatically deriving approximate circuits from a description of their (exact) functionality and from a notion of the allowed degree of inexactness. The first works in circuit approximation were the result of manual design, *i.e.*, approximate adders [65], [70], multipliers [10], [15], [22], [40] or dividers [16], [42] were created to design single inexact implementations of arithmetic units. Other works [20], [21] present algorithms that allow to automatically explore the energy-quality trade-off, but again limit the analysis to adders and multipliers only. Unlike these specific hand-crafted or circuit-specific designs, ALS aims instead at deriving approximate solutions for *any* circuit without *a priori* knowledge of its functionality.

The large family of approximate circuit design techniques can be divided into the two subcategories of Figure 1a: overscaling and *functional*. Overscaling aims at lowering a circuit supply voltage without reducing the corresponding operational frequency, thus reducing its static and dynamic energy while inducing timing errors. However, these timing errors may result in uncontrollably large computational errors, limiting the usability of these solutions [61] without redesign techniques, such as the ones proposed in [48], or careful considerations on the statistical distribution of the inputs [36].

In functional approximation – which is the focus of this survey – the function implemented by a circuit and/or the corresponding gate-level netlist is simplified, with the purpose of trading accuracy for performance.

This transformation of a generic Boolean function f into its approximate counterpart $\tilde{f}$ can be performed in different ways. We have identified three main categories for such approaches, illustrated in Figure 3: *Netlist transformation*, *Boolean rewriting* and *Approximate high-level synthesis*.

In **Netlist transformation** (Figure 3.a), the Boolean function f is already mapped into a netlist, *i.e.*, a list of electrical components connected together to form a circuit. For the same Boolean function, there exist several possibilities for netlist mapping. Many approaches belonging to this category, which will be described in detail in Section III, start from a gate-level netlist, whose components are Boolean gates implementing simple functions such as logical AND, OR and NOT. These methods transform such netlists by removing some nodes, or by substituting some wires with others, hence reducing the circuit size and power consumption.

Boolean rewriting approaches act on the function truth table, which represents a higher level of abstraction: no choice of employed electrical components has been made yet, only the list of possible inputs of f and the corresponding outputs is available. Therefore, this description is independent from the technology selected to map the function to a specific circuit. Methods belonging to this category, detailed in Section IV, modify the values of such outputs for a subset of the inputs, as illustrated in Figure 3.b: on the right column, some values in bold have been flipped w.r.t. the original truth table.

Finally, **Approximate high-level synthesis** focuses on the highest level of abstraction for ALS, where the function is described at behavioural level, such as in RTL Verilog or C language. An example fragment of such code is depicted in Figure 3.c, where a portion of C code shows how output values are computed from the function inputs by providing a mathematical expression, instead of listing all possible input combinations. These functions can be approximated as in the example, where the sum is transformed into a logical OR. Methods for approximate high-level synthesis are surveyed in Section V.

Regardless of the Approximate Logic Synthesis technique employed for simplification, the result is an approximate

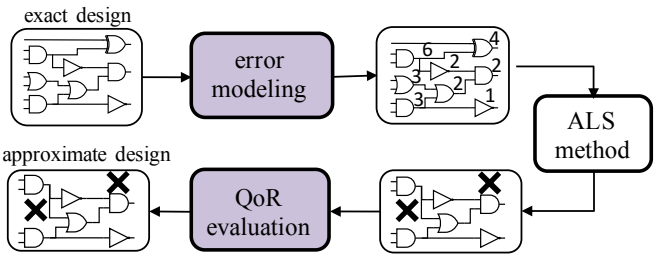


Fig. 4: A phase of *error modeling* can precede Approximate Logic Synthesis, with the aim of decorating a circuit (or a Boolean specification) with a notion of error, and hence guiding subsequent logic-simplification decisions. Then, a phase of *QoR evaluation* occurs to verify whether output quality constraints are satisfied in the synthesised approximate circuit.

version of the original f that will compute erroneous values for a subset of its inputs. If this error is limited, and can be tolerated by the application of interest, the original function can be replaced by its approximate version.

Section II of this paper is dedicated to error models and quantification for approximate circuits, since precise error estimation is a cornerstone in Approximate Computing. To appreciate the different phases in which error estimation is needed, Figure 4 illustrates a typical Approximate Logic Synthesis flow, where the ALS core method is preceded and followed by two distinct error quantification phases: *error modeling* and *QoR evaluation*.

Before applying a given approximation to our exact design, it can be useful to estimate how much that transformation will impact on the final result. Therefore, an *error modeling* phase can be present, with the aim of annotating a circuit (or a Boolean) specification with a notion of error, as depicted in the top row of Figure 4. This step provides an estimate – which can be more or less accurate, depending on the approach – of the potential error introduced by a given circuit simplification, which in turn can *guide* ALS methods in identifying the least error-prone transformation – or set of transformations.

Then, a phase of *QoR evaluation* occurs once the circuit logic has been modified and simplified by an ALS algorithm – as shown in Figure 4 – to verify whether output quality constraints are satisfied in the synthesized approximate circuit.

Note that not all ALS methods reviewed in Sections III, IV and V rely on *both* error modeling and QoR evaluation phases. Section II provides examples and describes methods that undergo one, both, or no such phases.

The organization of this paper is as follows: Section II, along with a formal definition of the most commonly employed error metrics, further extends the description of the error quantification phases, and summarizes notable works on accurate error modeling. Sections III, IV and V describe the strategies for approximate hardware synthesis depicted in Figure 3. In particular, Section V discusses methods that derive approximate hardware components directly from high-level source code. Finally, Section VI compares the performance of different works previously presented, providing insights on their efficiency and their characteristics in addressing ALS problems.

II. Methods for Error Estimation

As introduced in Section I, key to ALS is the evaluation of the *error* induced by a simplification, which can be for instance the removal of a gate, or the modification of a value in a truth table. Hence, as shown in Figure 4, ALS methods can be preceded by an error modeling phase and followed by a QoR evaluation phase. These two phases are not necessarily present in all ALS methods.

For example, Vasicek *et al*. [54] employ a genetic programming technique that does not need the guide of *a priori* error modeling, since it automatically evolves towards lower-error circuits, while other works, such as Scarabottolo *et al*. [44], compute a tight bound on maximum error in the first phase, and then the resulting simplified circuits do not need to be re-evaluated as they are already guaranteed to not overtake such bound.

In Section II-A we review the error metrics that can be of interest when designing approximate circuits, and in Section II-B we review the state of the art in methods used for error modeling.

A. Error Metrics

When performing error profiling, a first step is to appropriately encode the bits at the output according to the intended representation (*e.g.*, as signed or unsigned numbers). Hence, a difference d between an exact and approximate implemented Boolean functions (f and $\tilde{f}$, respectively) can be computed between two outputs for the same inputs:

$$d(f(x), \tilde{f}(x)) = ||f(x) - \tilde{f}(x)||$$

Then, an input-independent distance D must be derived from all values of d according to a metric. Alternative choices for such metric are influenced by several factors: the nature of the application in which the approximate hardware will be employed, its criticality, etc. For example, Ma *et al.* [30] and Venkataramani *et al.* [57] employ the Hamming Distance as a measure of d, defined as the number of bits flips in $\tilde{f}$ w.r.t. the original f.

We will now define the most common, widespread metrics for D employed in the field. Referring again to the Hamming distance, one could be interested in the maximum or average Hamming distance over the inputs of f.

When, as done in [5], [43], [44], [45], the focus is on controlling the Maximum Error (*i.e.*, worst case distance), that occurs when a circuit is approximated, D is defined as:

$$\max_{x \in X}(d(f(x), \tilde{f}(x)))$$

expressing the maximum value of the difference between f and $\tilde{f}$, being X the set of all possible circuit inputs and x a generic input.

Several Approximate Logic Synthesis techniques [19], [26], [27], [45], [56], [60], [64] monitor average case distance (mean

absolute error) induced on the output, instead of focusing on potential outliers, expressed as

$$\mathbb{E}_x\{d(f(x), \tilde{f}(x))\}$$

where the expectation is taken over the input data. If inputs are uniformly distributed, the expression above becomes

$$\frac{1}{|X|}\sum_{x\in X} d(f(x), \tilde{f}(x))$$

, where $|X|$ denotes the input set cardinality (*i.e.* the number of possible inputs). A related metric is the Mean Squared Error, in which distance terms are squared:

$$\frac{1}{|X|}\sum_{x\in X} (d(f(x), \tilde{f}(x)))^2$$

Distances are instead normalised by the (exact) output values size $||f||$ when considering the Average Relative Error Magnitude as an error metric:

$$\frac{1}{|X|}\sum_{x\in X} \frac{d(f(x), \tilde{f}(x))}{||f(x)||}$$

Moreover, it is usually of interest to know *how often* errors occur in approximate circuits, regardless of their magnitude. The Error Rate is a common metric that captures this phenomenon. For a generic circuit, given $W = \{x \in X | f(x) \neq \tilde{f}(x)\}$ the set of inputs for which the approximate function computes an erroneous output, the Error rate is defined as:

$$\frac{|W|}{|X|}$$

It is of course possible for a given Approximate Logic Synthesis approach to consider more than one error metric. Indeed, in [5], [31], [39], [58], [49], [57], [51] both maximum error and average error are taken into account.

Some works introduce further metrics for error estimation that also take into account the structure of the circuit being simplified, in addition to its functionality. Notably, Zhang *et al*. [69] adopt the notion of Approximate Efficiency, defined as the ratio between the gain (in terms of Energy-Delay Product, EDP) deriving from the simplification of a node and the corresponding induced error: $\Delta EDP/D$. The underlying assumption is that, if two nodes generate the same error in the final output when pruned from the original circuit, the one leading to higher benefits should be pruned first.

More abstract QoR metrics are usually employed by Approximate High Level Synthesis (AHLS) frameworks, focus of Section V. AHLS designs employ inexact arithmetic circuits as building blocks to realize approximate accelerators. Approximation-induced errors are therefore usually expressed by domain-specific indicators, retrieved a-posteriori in the QoR evaluation phase via simulation over a suite of representative inputs. Signal-to-Noise Ratio (SNR) is evaluated for signal processing designs [25] [24], while Structural Similarity (SSIM, [67]) is used in image processing ones [34]. Furthermore, classification accuracy assesses QoR in [20] [38]. In [2], all three of these metrics are considered, in accordance with the domain of each of the considered benchmarks.

B. Methods for Error Modeling and for QoR Evaluation

For all metrics introduced above, a precise computation of the error caused by a circuit modification requires exhaustive evaluation of all possible input combinations. This number grows exponentially with the input precision, and hence such computation cannot scale to large circuits. SAT-solver based techniques were proposed [58] in order to help accelerating exhaustive evaluations; however, exhaustiveness necessarily becomes intractable at some point, as the circuit size increases. Hence, the need arises for methods to efficiently calculate error estimates (in the case of average errors and error rates) and error bounds (for maximum errors). In the following, we review the body of work for such methods.

Average error estimation. A widespread strategy to estimate average errors is to simulate a circuit for a *subset* of its inputs, randomly chosen through Monte Carlo selection [58], resulting in an unbiased statistical estimate of D. Nonetheless, even a Monte Carlo implementation can become computationally intractable for large circuits if carried out in a straightforward way, because it necessitates distinct evaluations, at each simplification step, for all candidate approximate transformations.

Su *et al*. [53] devise a technique that effectively lowers the computational effort entailed. Their strategy, illustrated in Figure 5, aims at estimating the error introduced by a set of candidate approximate transformations (ATs) without having to resort to Monte Carlo simulation for each of them. They propose a two-step method that computes a three-dimensional 0-1 change propagation matrix (CPM), then performs batch error estimation for all candidate transformations in a single Monte Carlo run. CPM has size $M \times N \times O$, where M is the number of Monte Carlo random input patterns, N the number of nodes in the netlist being simplified and O the cardinality of its output. An entry in CPM $[i, n, o]$ is 1 if and only if a change on the node n propagates to the output o under the i-th pattern. The CPM is computed in a reverse topological traverse of the the netlist, starting from the primary outputs, then recursively calculating the entries for each node fanin.

The same CPM can then be employed to evaluate, within a single Monte Carlo pass, the error rate and the average error magnitude derived from all candidate transformations. The batch calculation simply computes the desired error metric for each AT individually, but thanks to the CPM, there is no need to re-run Monte Carlo simulation at each step, since information on the error propagation towards the output is retained in the matrix and the desired metric is computed cumulatively.

Once a transformation is selected, a new matrix CPM is calculated, and the methodology iteratively proceeds by evaluating candidate transformations on the newly-derived inexact circuit with respect to the exact counterpart, until an average error / error rate constraint is violated. The computational complexity of this strategy is $\mathcal{O}(MOT)$, where T is the size of the set of possible approximate transformations ATs, compared to $\mathcal{O}(MNT)$ for a naïve Monte Carlo alternative. Note that the number of outputs O of a circuit is usually much smaller than the number of nodes N.

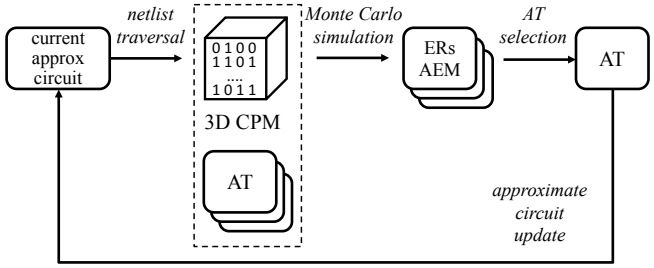


Fig. 5: Block scheme of the batch error estimation technique in [53], where at each iteration a set of candidate Approximate Tranformations (AT) is evaluated through the Change Propagation Matrix (CPM) to obtain the corresponding Error Rate (ER) and Average Error Magnitude (AEM).

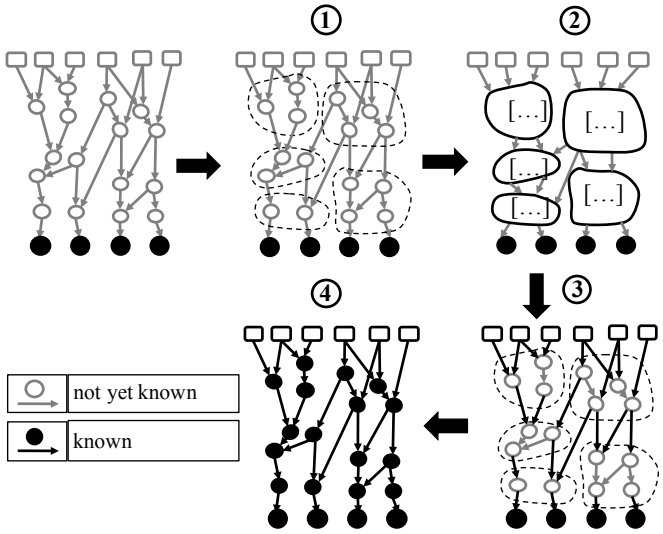


Fig. 6: The four steps in the Partition and Propagate methodology., from [43]: 1) partitioning, 2) derivation of the propagation matrices, 3) computation of the weights across partitions and 4) subgraph simulations.

Bounding maximum errors. Monte Carlo-based approaches are not employable when maximum error thresholds must be provided, as they can not account for outliers. To compute error guarantees, Schlachter *et al*. [45] introduced an algorithm which assigns to each node in a circuit the sum of the significance of all its reachable outputs (where the significance of the output bit i is equal to 2^i). This strategy is overly conservative, as it assumes that a node simplification can affect all reachable outputs simultaneously for at least one input combination (all ones becoming zeros, and vice-versa). In fact, masking effects very often reduce the magnitude of perturbations caused by inexact transformations, preventing all outputs to assume erroneous values at the same time.

Tighter maximum-error bounds are derived in a recent work by Scarabottolo *et al*. [43]. The goal of their *Partition and Propagate* (P&P) methodology is to fully label a circuit, *i.e.* to assign weights to each node corresponding to a bound on the maximum difference from the exact output if such node is removed and its output set to a constant value. The weight of a node i is hence

$$w(i) \geq \max_{x \in X} |f(x) - \tilde{f}_i(x)|$$

where $\tilde{f}_i$ is the circuit functionality when gate i is removed.

As illustrated in Figure 6, weights are computed by P&P in four steps:

1) *Partitioning:* The combinational part of the circuit, represented as a direct acyclic graph, is divided in subgraphs, so that each subgraph has less than I_s inputs (where I_s is much lower than the number of primary inputs of the graph).
2) *Derivation of propagation matrices:* Since the input size of subgraphs I_s is small, it is computationally feasible, for all combinations of inputs, to express the weights of the subgraph inputs as a function of those of its outputs. P&P does so by observing the sub-graphs truth tables, deriving their propagation matrices M. The relation between a subgraph input weights vector $\mathbf{w}_{in}$ and output weights vector $\mathbf{w}_{out}$ is then
$$\mathbf{w}_{in} = M\mathbf{w}_{out}$$
3) *Propagation:* Weights are then propagated across subgraphs considering them in reverse topological order. If the children nodes of a subgraph output belong to different subgraphs, the subgraph output node weight $\mathbf{w}_{out}$ is conservatively set as the sum of the $\mathbf{w}_{in}$ elements pertaining to the successor subgraphs.
4) *Subgraph simulations:* Finally, the weights of nodes inside subgraphs are retrieved using exhaustive simulation *separately for each subgraph*. Again, this is feasible since the number of subgraph inputs is limited.

The computational complexity of P&P is $\mathcal{O}(N + S + E)$, where N is the number of circuit nodes, S the number of subgraphs and E the number of edges traversing distinct subgraphs.

III. ALS: Structural Netlist Transformations

Among methods that implement structural netlist transformation, we review five different works, which we group according to their adopted strategy:

A. Greedy Heuristics for Netlist Pruning;
B. Greedy Heuristics for Netlist Manipulation;
C. Stochastic Netlist Transformation; and
D. Exhaustive Exploration for Netlist Pruning.

A. Greedy Heuristics for Netlist Pruning

Shin *et al*. [49] employ a greedy strategy for generic circuit simplification, applied to adders used in image compression and decompression. In their methodology, a set of multiple stuck-at-faults (SAFs) is identified, and these SAFs are injected in the original circuit by assigning a static 0 or a static 1 to each signal of a selected SA0 or SA1 fault.

Two procedures for circuit simplifications are then applied:

1) *Backward simplification*: this operation traverses the circuit from the SAF node towards the primary inputs, marks

all nodes whose fanout is now empty as deletable, and eliminates marked nodes;

2) *Forward simplification*: performs the same operation in the forward direction, traversing the SAF fanouts towards the primary outputs. In this second step, however, the type of SAF (0 or 1) coupled with the logic functionality is exploited for further reducing the logic stemming from the given node.

A greedy heuristic is employed to iteratively choose the SAF that maximizes a given figure of merit (*e.g.*, area reduction), simplify the circuit forward and backward, and repeat the process until the error constraint is violated. A parallel fault simulator with a set of test vectors is used to evaluate the error on the final output at each SAF simplification.

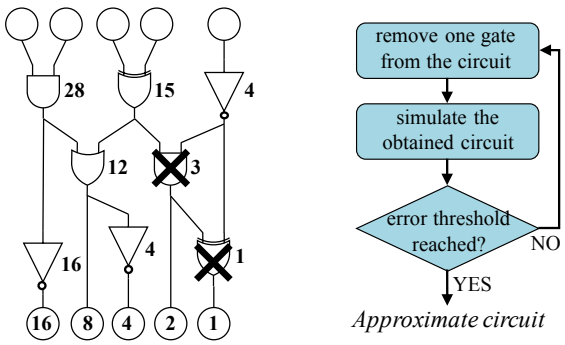


Fig. 7: GLP [45] framework. Gates are removed from a generic circuit starting from the least significant, until the allowed error threshold is reached.

GLP by Schlachter *et al*. [45] presents another greedy iterative algorithm for circuit simplification. The proposed framework, depicted in Figure 7, is simple but effective. The exact circuit is represented as a direct acyclic graph and nodes are pruned according to two main criteria: the node significance, which represents the impact of that node on the final output, and the node activity or toggle count. According to the application characteristics, nodes can be pruned starting from those with lower significance, lower activity, or a combination of the two: the significance-activity product (SAP). Node activity is obtained through gate-level hardware simulation, while significance is computed in a reverse topological graph traversal, as mentioned in Section II-B, starting from the primary outputs arithmetic bit-significance, then assigning to each node i the significance σ_i:

$$\sigma_i = \sum \sigma_{desc(i)},$$

where $desc(i)$ are all direct descendants of node i.

After nodes have been ranked according to the desired metric, the GLP framework iteratively removes a node from the original circuit, setting its output to a constant; it then resynthesizes the circuit, simulates it with a Monte Carlo process to verify that the error constraints on error rate and mean relative error have not been violated, and recomputes SAP for node ranking. When the error threshold is reached, the algorithm stops.

Computing node activity can take a considerable amount of time (15 to 20 minutes for a 32-bit adder with 5 million input combinations). However, it allows the selection between a wider range of performance-accuracy trade-offs for the same amount of tolerated error. Therefore, significance-only node ranking is preferred for a first, fast design, while SAP ranking can be employed for fine tuning.

B. Greedy Heuristics for Netlist Manipulation

Venkataramani *et al*. [56] propose another greedy strategy called SASIMI (Substitute-And-SIMplIfy). In SASIMI, functional approximation is performed by identifying pairs of signals that assume the same value with high probability, and substitute one with the other. The authors call TS (target signal) the signal to be replaced, and the one employed at its place SS (substitute signal). SS can be a constant value (a logic 1 or a logic 0), or another signal of the original circuit.

The key idea of the paper is illustrated in Figure 8, where TS is substituted by SS. When the target signal is replaced, the gates belonging exclusively to its generating cone of logic are removed from the circuit. Moreover, the logic in TS fanout can potentially be reduced, as well as that belonging to TS fanin *and* to other signals' fanin. Therefore, both direct pruning and indirect downsizing are considered in the choice of TS.

Clearly, the error induced by a potential substitution must be considered in the choice of TS and SS. This error can be estimated by analyzing the *difference signal*, expressed as XOR of TS and SS, and its probability P_{DIFF}, which indicates the probability of TS being different from SS. Since a signal's complement is a possible candidate for substitution too, the preferred signal is the one with smallest probability product $P_{DIFF}(1 - P_{DIFF})$.

However, the difference probability does not necessarily reflect the error that will be introduced to the final circuit output, hence this has to be evaluated separately, through a Monte Carlo process that estimates the error rate and average absolute error magnitude on a subset of all possible circuit inputs, which are assumed to be uniformly distributed. The algorithm takes as input the original circuit and a target error, then iteratively performs the selection of the best candidate signal pair, the substitution and consequent circuit simplification, followed by QoR evaluation. Once the target error constraint is reached, the iterative algorithm stops.

The authors couple the ALS algorithm described above with an adaptation for quality configurable circuits, where the gates in TS's cone of logic are not eliminated, and an additional circuit is employed to monitor the difference between TS and SS. Based on the desired quality, the discrepancy between these two signals can either be ignored or recovered. Although quality monitoring and error recovery introduce a substantial overhead, the experiments show that this method leads to significant energy savings (although lower than those obtained in the approximate mode).

C. Stochastic Netlist Transformation

Liu *et al*. [27] argue that the assumption of uniform distribution of input data is seldom correct. Therefore, they propose

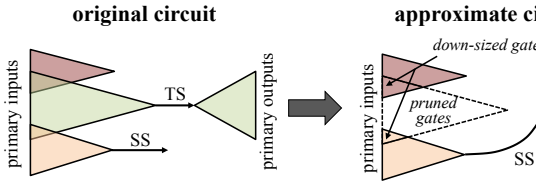


Fig. 8: SASIMI [56] illustration of signal substitu
TS is replaced by SS, gates belonging exclusively t
cone are deleted, while those shared with other
downsized.

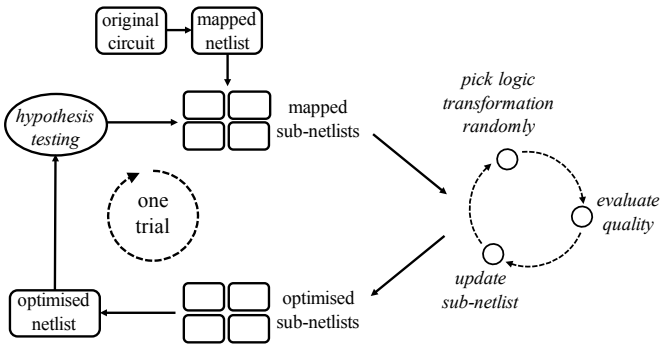


Fig. 9: The nested-loops approach of SCALS [27]. For each trial, all mapped sub-netlists are optimised in parallel through iterative random selection of possible transformations. Once the optimised sub-netlists are recomposed, the error of the circuit is assessed through hypothesis testing.

SCALS: Statistically Certified Approximate Logic
an iterative framework which presents some radical
with respect to those described in the previous s
First of all, the authors of SCALS apply statistical
testing to estimate the errors obtained on the circuit outputs after an approximate transformation, to guarantee that the population behaviour is indeed a faithful representation of the actual data distribution.

Secondly, their algorithm presents two nested iterative loops, as depicted in Figure 9: the original circuit is immediately mapped to the desired technology, in order to work directly on what will be the actual implementation. The mapped netlist is then partitioned into sub-netlists, which will be independently optimised, in parallel, through the inner loop on the right of Figure 9.

The resulting optimised sub-netlists are then recombined, and the resulting netlist error is evaluated through statistical hypothesis testing. Such sequence of operations represents a single trial, and the process continues until the desired error constraint is reached. Therefore, they are able to provide a confidence level for the analyzed error metrics, namely error rate and average relative error magnitude.

It is interesting to understand how the sub-netlists optimisation (the inner loop) works: SCALS considers a set of logic transformations $T = E \cup A$, where $E = \{\text{BALANCE, REWRITE, REFACTOR}\}$ represents exact transformations, and $A = \{\text{REDUCE, FLIP, ADD}\}$ approximate ones. At each iteration, a transformation i is selected from T with probability p_i.

While exact transformations do not modify the circuit functionality, but may optimize its implementation, approximate transformations do alter it. REDUCE and FLIP randomly pick a logic gate from the netlist, the first removes one of its fanins, whereas the second inverts its output. An ADD transformation instead inserts a two-input logic gate to the circuit, connecting it to existing signals at random.

For each logic transformation, the approximate sub-netlist is mapped to the desired technology to assess its quality. Statistical inference techniques are used to estimate the errors introduced in this phase, since hypothesis testing for each transformation would be computationally infeasible. If the quality improves, the sub-netlists are updated and the process starts over until a fixed point is reached.

In SCALS, the transformation space is explored stochastically, which means considering possible transformations at random, instead of employing fixed heuristics, so as to maximize the number of design points tackled.

In a similar direction, Vasicek and Sekanina propose EvoApprox, a genetic algorithm to mutate the circuit into approximate versions by swapping gates with wire connections [54]. Circuits are represented as direct acyclic graphs, whose nodes can be Boolean gates or more complex components according to the technology library chosen. The nodes are contained in a two-dimensions grid, the *chromosome*, which is randomly modified to explore new design points. This mutation evolves using a fitness function, which leads to better approximation over runtime. After computing area and error of the initial population, the algorithm iteratively selects the best-scored circuit, generates λ offspring from the parent through mutation, and evaluates the new population. The authors develop three versions of the genetic algorithm:

- *Resource-oriented method*: the error is minimized under the assumption that only a fraction of the components (gates) needed to implement the accurate circuit is available;
- *Error-oriented method*: the area is minimized while keeping the error between a user-specified range;
- *Multi-objective*: this algorithm version allows to optimize the error and other circuit parameters, such as area, power consumption and delay.

To evaluate the error obtained at each mutation, full-simulation is employed for small circuits, while for larger circuits the authors resort to more complex techniques such as SAT or BDD-based evaluation.

D. Exhaustive Exploration for Netlist Pruning

As opposed to other works surveyed in this section, Circuit Carving by Scarabottolo *et al.* [44], does not employ iterative approximations towards inexact logic synthesis. It instead resorts to exhaustive exploration of all possible nodes subsets that can be removed from the exact circuit, among which the most convenient will be chosen. The best candidate sub-circuit

is the largest one (in terms of number of gates) that does not overcome the identified error threshold.

The key idea of this approach is to consider the effect of multiple pruning choices combined, eliminating the risk of getting stuck in local minima. However, the efficiency of the algorithm strongly relies on accurate estimation of node significance, either through exhaustive simulation (if the circuit size allows it), or by exploiting the circuit regularity to derive node significance through induction.

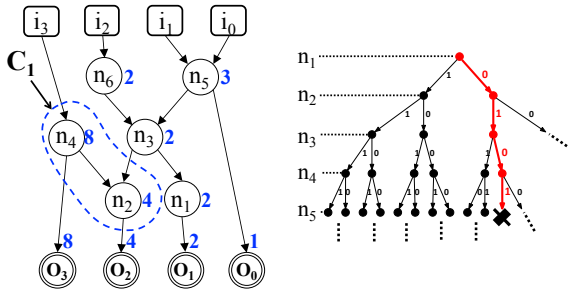
Once all nodes are labelled with their significance, a binary tree search algorithm explores all possible subsets, called *cuts*, and estimates the error induced on the output by the removal of the whole cut. Figure 10a illustrates an example of a cut, enclosed in the dashed blue line. Each node in the graph is labelled with its significance. The cut significance is defined as the sum of all its output nodes significance: in the example, $\sigma_{C_1} = \sigma_{n_4} + \sigma_{n_2} = 8 + 4 = 12$.

Figure 10b shows how the binary tree search algorithm works: each level in the exploration tree represents the inclusion (or exclusion) of a given node in the cut. For a graph with *N* nodes, 2^N branches represent all possible nodes subsets that can be considered as best candidates for approximation. The red highlighted path corresponds to cut C_1 of Figure 10a: indeed, 1-branch is taken for node n_2 and n_4 only. The algorithm identifies the largest cut whose significance does not overcome the pre-defined error threshold T.

Since the worst-case complexity of the exploration is exponential in the number of nodes, three criteria are employed to bound it and strongly reduce the average complexity:

1) *Validity*: if the cut significance overcomes the available error threshold T, the cut is not valid and the exploration stops;
2) *Closure*: a cut is defined *closed* when, for all its nodes, if all children of a node n belong to the cut, n is also in the cut. The same holds for parents: if all parents of a node n are in the cut, n is in the cut as well. This means that there is no larger cut containing the current one with the same significance. Only closed cuts are considered as possible solution: if a branch represents a non-closed cut, it is abandoned.
3) *Residual gain*: if, at some level in the exploration, the sum of the nodes still to be considered plus the nodes already included in the cut is less than the size of the best candidate already found, the algorithm avoids exploring any further, since there could not be any additional advantage in exploring lower levels.

These criteria for bounding exploration prove to be effective in reducing by orders of magnitude the number of points explored. However, for large designs, the complexity is still too high to be treated in a reasonable amount of time. Even when the exploration is stopped after having reached a time limit, this method performs better than GLP by Schlachter *et al.*, to which it is compared, in terms of energy, delay and area reduction of the approximate designs for the same tolerated error. The advantage in performance over the chosen greedy algorithm is explained by two factors: first, the exhaustive exploration leads to optimal solutions that are often overlooked by greedy strategies; moreover, the error estimation of single gates is much more accurate, hence guiding the exploration towards better candidates for Approximate Logic Synthesis.



(a) (b)

Fig. 10: (a) An example of cut for Circuit Carving [44], with significance given by its output nodes, and (b) the corresponding path in the binary tree search, where the 1-branch is taken for node n_2 and n_4 only.

IV. ALS: Logic Rewriting Based Methods

We categorize ALS using *Boolean rewriting* as a general approach in which the logic of the circuit is first captured in a formal Boolean representation that is manipulated to yield an approximate Boolean representation; this is, in turn, synthesized to a gate-based netlist. We review four different techniques for this general approach:

A. logic rewriting by Boolean optimization;
B. logic rewriting by Boolean matrix factorization;
C. logic rewriting by binary decision diagrams; and
D. logic rewriting by and-inverter graphs.

A. Logic Rewriting by Boolean Optimization

In this approach, the approximations are captured into Boolean expressions that are used to relax the Boolean minimization of the original circuit, leading to an approximate circuit of smaller size [31], [57], [60]. One of the earliest works to employ this approach is SALSA [57]. In SALSA a *QoR circuit* is first constructed by comparing the outputs of the original circuit and the approximate circuit using a comparator, as illustrated in Figure 11. Depending on the error metric, the QoR circuit can compute the arithmetic difference or the Hamming distance (HD) between the outputs of the original and approximate circuits. If the error metric is the arithmetic difference and the outputs of the original circuit and the approximate circuit are denoted by $f(i_1, \ldots, i_n)$ and $\tilde{f}(i_1, \ldots, i_n)$ respectively, then the QoR circuit is given by

$$QoR = 1 \text{ if and only if } |f(i_1, \ldots, i_n) - \tilde{f}(i_1, \ldots, i_n)| \leq B,$$

where B is the maximum error bound and QoR is a binary output. Any acceptable approximate circuit, $\tilde{f}$, will have a tautology output for the QoR circuit; however, an excessive approximation in $\tilde{f}$ will lead to some input combinations that cause $QoR = 0$. Thus, one is free to simplify the logic of $\tilde{f}$ as long as the output of the QoR circuit stays as a tautology.

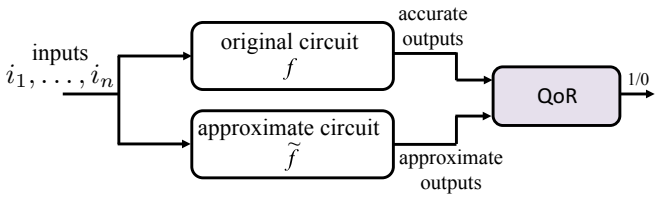


Fig. 11: In SALSA a QoR circuit is constructed to compare the outputs of exact and approximate circuits [57]. Observability *don't cares* of the approximate circuit are used to minimize the approximate circuit logic.

To simplify the logic of the approximate circuit, SALSA computes the *observability don't cares* for each one of the outputs of the approximate circuit with respect to the primary outputs of the QoR circuit. For each output of the approximate circuit, these *don't cares* are the set of primary input combinations for which the outputs of the QoR circuit are insensitive to the output of the approximate circuit. This set of *don't cares* can be then used to minimize the approximate circuit using standard logic synthesis techniques [46].

For example, consider f to be a circuit with three inputs $\{i_1, i_2, i_3\}$ that computes two outputs: $f_1(i_1, i_2, i_3)$ and $f_2(i_1, i_2, i_3)$, and a QoR circuit that computes the HD between the outputs of the circuits and its approximate counterpart. The QoR circuit outputs a 1 as long as the HD between the original and approximate circuit is less than or equal to one. The observability *don't care* set for the approximate output $\tilde{f}_1$ is equal to the input combinations that lead to $f_2 = 1$ and $\tilde{f}_2 = 1$ or $f_2 = 0$ and $\tilde{f}_2 = 0$, which can be expressed in terms of the primary inputs as

$$(f_2(i_1, i_2, i_3) \wedge \tilde{f}_2(i_1, i_2, i_3)) \vee (\overline{f_2}(i_1, i_2, i_3) \wedge \overline{\tilde{f}_2}(i_1, i_2, i_3)).$$

This set of *don't cares* can be used to minimize the entire logic of $\tilde{f}_1$ using standard *don't care* logic minimization techniques [46]. SALSA has been extended in ASLAN [39] to handle sequential circuits, where errors arise over multiple sequential cycles. ASLAN uses a circuit block exploration method to identify the impact of approximating the combinational blocks, and then uses a gradient-descent approach to find good approximations for the entire circuit.

In contrast to SALSA's global minimization approach, Wu and Qian [60] propose a local minimization approach to simplify a circuit by substituting the Boolean expressions of the internal circuit nodes with approximated expressions that require less logic. For instance, consider an internal circuit node that computes the logic $(a \vee b) \wedge (c \vee d)$, which can be re-written in an approximate way by dropping one literal, leading to four possible choices: $a \wedge (c \vee d)$, $b \wedge (c \vee d)$, $(a \vee b) \wedge c$, or $(a \vee b) \wedge d$. Each of these possibilities has different impact on error and circuit complexity, as measured by design area. Since a circuit can have million of internal expressions, each with a number of possible ways to re-write approximately, a procedure is needed to identify the best expressions to apply the simplifications, together with the particular form of simplification that leads to minimal area. Each expression is given both a *value*, defined as the reduction in area it realizes when simplified, and a *weight*, defined as the introduced error, then a knapsack formulation is constructed and solved to identify the best set of nodes to approximate in order to maximize value (*i.e.*, total area reduction) under weight constraints (*i.e.*, maximum error).

B. Logic Rewriting Using Boolean Matrix Factorization.

BLASYS introduces a new formal method in approximate logic synthesis [17], [19], [30]. In BLASYS the operation of a circuit is captured by a matrix that represents the output side of the circuit's truth table, such that for a n-input, m-output circuit, the matrix size is $N \times p$, where $N = 2^n$. To create an approximate circuit from a given circuit, Boolean matrix factorization is used, where an input matrix M of dimensions $N \times p$ is factored into two matrices: an $N \times d$ matrix, B, and an $d \times p$ matrix, C, where $1 \leq d < p$ such that $|M - BC|_2$ is minimized; *i.e.*, $M \approx BC$ [32]. In Boolean Matrix Factorization (BMF), multiplications are performed using logical AND operations and additions are performed using logical OR operations. One can interpret the columns of B as *factors* or *bases* that are linearly combined using C. For example, consider a circuit with $n = 3$ inputs and $p = 4$ outputs and the truth table M given in Equation (IV-B)[2]:

$$M = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Then for the case of $d = 2$, M can be factored as given in Equation (1):

$$M \approx \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 0 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & \underline{0} \\ 1 & 1 & \underline{0} & \underline{0} \\ 0 & 1 & \underline{0} & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & \underline{0} \\ 1 & \underline{1} & 0 & \underline{0} \end{pmatrix}$$

leading to errors in the approximation, which is equal to 4 as measured by the Hamming Distance (HD) to the original matrix, or a relative HD error of $4/32 = 12.5\%$. The factorization degree, d, controls the amount of approximation in the factored representation, with the general trend that reducing d increases the amount of approximation.

After the matrix representing the circuit is factored, a new approximate circuit is created by synthesizing (1) the circuit representing the matrix B, which is referred to as the *compressor circuit*, and (2) the circuit representing the matrix C, which is referred to as the *decompressor circuit*, wherein C operates on the outputs of B by ORing them. Since the compressor has fewer outputs than the original circuit, it

[2]The order of the rows in M corresponds to inputs $000, 001, \ldots, 111$.

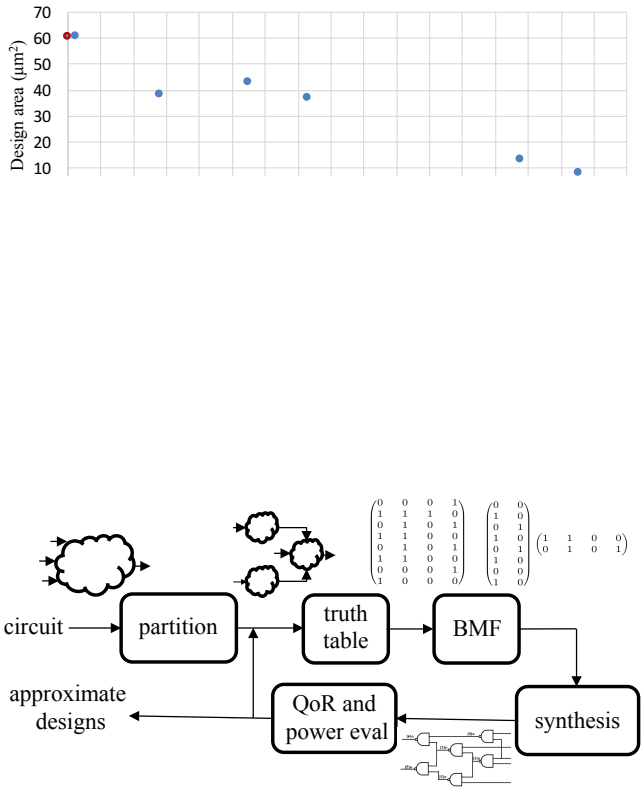


Fig. 13: BLASYS flow [30]. An input circuit is first partitioned into subcircuits with a reasonable number of inputs for each subcircuit. The truth table of every subcircuit is then evaluated followed by Boolean Matrix Factorization (BMF). The results from BMF is then synthesized to create an approximate subcircuit. The QoR and physical metrics (*e.g.* power or area) of the entire circuit are then evaluated. This process is iterated multiple times to yield approximate circuits with various QoR-power trade-offs.

typically leads to a synthesized circuit with less design area and power consumption. Figure 12 gives an example of the trade-off between QoR and design area where BLASYS is used to synthesize approximate versions for circuit `x2` from the LGSynth91 benchmark set [63]. `x2` has 10 primary inputs and 7 primary outputs; thus, one can vary the factorization degree, d, from $d = 6$ down to $d = 1$. By changing d, BLASYS enables a graceful trade-off between QoR and design area. For example, for `x2` we can reduce design area by 36% at the expense of introducing 2.79% bit flips in the truth table.

Since the construction of the matrix that represents the truth table is limited by the number of primary inputs of the circuit, BLASYS incorporates a hypergraph partitioning method that breaks down a large circuit into a number of subcircuits as illustrated in Figure 13, such that BMF can be applied to each subcircuit to yield an approximate subcircuit. The original subcircuit is substituted by its approximate subcircuit and then the QoR and design area or power are evaluated for the entire circuit. BLASYS uses a design space exploration method to identify the best subcircuits to approximate together with the factorization degree for each subcircuit.

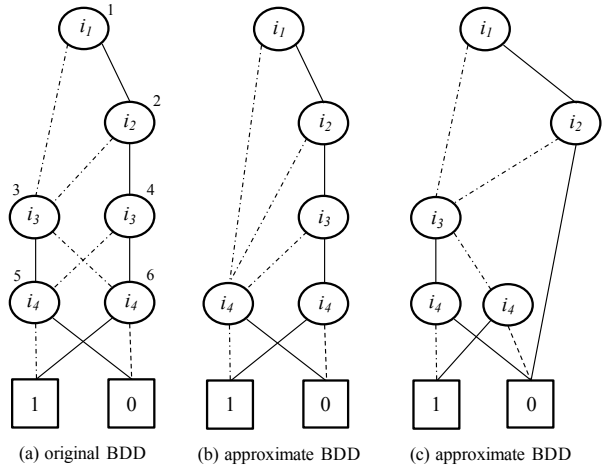


Fig. 14: Examples of approximate Boolean re-writing using BDDs.

C. *Logic Rewriting Using Binary Decision Diagrams.*

Reduced Order Binary Decision Diagrams (BDDs for short) are a canonical representation for Boolean circuits [4]. In a BDD, internal nodes represent the primary inputs and edges represent a possible assignment for a primary input, *i.e.* 1 or 0. A BDD has two terminal nodes corresponding to the two possible circuit outputs (0 and 1), and a path from the root of the BDD to a terminal represents the output that will result from an assignment to the primary inputs along the path. Figure 14a gives an example BDD for a circuit with four inputs, i_1, i_2, i_3 and i_4 and a single output. Based on this BDD, an assignment for nodes 1, 3 and 5 corresponding to $i_1 = 0$, $i_3 = 1$, and $i_4 = 0$ will result in a circuit output of 1 (irrespective of the value of i_2). The number of nodes in a BDD, *i.e.*, its *size*, is sensitive to the order of variables in the BDD. Minimizing the size of the BDD reduces the size of the circuit synthesized from the BDD [62].

The general idea of BDD-based approximate synthesis approaches is to first represent an input circuit as a BDD, and then transform it to produce an approximate BDD with reduced size. The approximate BDD is accepted and synthesized to a circuit as long as the error between the original BDD and approximate BDD does not exceed a given error bound [13], [51]. One possible transformation is to replace a node with one of its children, *i.e.*, *co-factors*. For example, the BDD in Figure 14b is obtained from the BDD in Figure 14a by replacing node 3 with one of its children (node 5). Another possible transformation is *rounding*, where a child of a node is either replaced by the terminal 1 or the terminal 0. Since each internal node has two children, the *lighter child*, *i.e.*, the child with fewer number of assignments that lead to the terminal 1, is chosen. For example, the approximate BDD in Figure 14c is obtained from the BDD in Figure 14a by replacing one of the children of node 2 by the terminal 0.

The aforementioned approach does not guarantee minimum BDD size for a target error bound. It is possible to devise a method to identify the minimum BDD for a given error bound, where the output of this function differs in at most

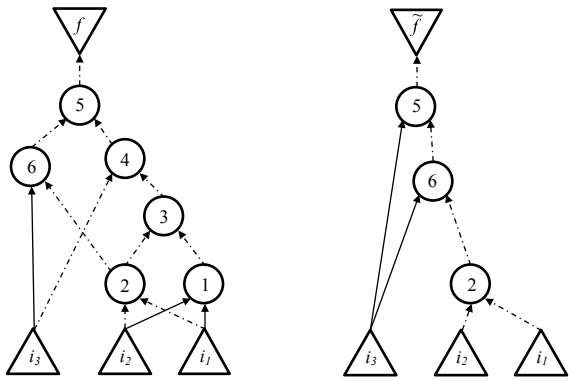
e possible input combinations from the origina
To identify this function a new BDD, g, is con
that it enumerates every possible function whose
in at most e bits compared to the original circu
represents every potential circuit approximation
e output flips. For example, consider a single-c
$f(i_1, i_2)$, with two primary inputs, (i_1, i_2), the
output possibilities for each one of the possib
combinations $\{i_1 i_2 = 00, i_1 i_2 = 01, i_1 i_2 = 10, i$
new BDD $g(b_1, b_2, b_3, b_4, i_1, i_2)$ is constructed w
tional variables $\{b_1, \ldots, b_4\}$ that are created to i
input combinations will have an output bit flip;
for $j \in \{1, \ldots, 4\}$ then $g(b_1, b_2, b_3, b_4, i_1, i_2)$
otherwise, $g(b_1, b_2, b_3, b_4, i_1, i_2) = f(i_1, i_2)$. In
g, a partial path that assigns values for b_1, b_2, b_3
to 0 if the output function of the approximat
more than e bit flips; otherwise, it will lead t
BDD, whose variables are the two original c
This subgraph BDD represents the logic of the approximate circuit corresponding to a particular set of output bit flips, as determined by the assignment of $\{b_1, \ldots, b_4\}$ in the partial path. If the paths of this subgraph BDD are enumerated and compared to the original BDD, we will find that no more than e paths lead to different outcomes.

D. Logic Rewriting Using And-Inverter-Graphs (AIGs).

An AIG is a graph representation for circuits where nodes correspond to two-input AND gates and edges can be either inverted or not inverted [33]. For example, Figure 15 shows circuits represented in AIGs where dashed edges represent inverted signals, and where i_1, i_2 and i_3 represent the primary input signals and f the primary output signal. AIGs provide a scalable graph representation for circuit synthesis; however, unlike BDDs they are not canonical. Using AIGs as the Boolean representation, Chandrasekharan *et al.* [5] propose an algorithm for approximate AIG re-writing that guarantees the bounds of approximation errors introduced in the approximate circuit. First, the critical path(s) are identified in the AIG, where the critical path(s) are the paths from the primary inputs to the primary outputs with the largest number of nodes. For example, in Figure 15a, the path $\{1, 3, 4, 5\}$ is the critical path. After identifying the critical path(s) in the AIG, cut enumeration on the selected paths is used to identify potential cuts. In Figure 15a, nodes $\{1, 3\}$ represent a cut of size 2. A cut can be replaced by an approximation for it; a simple approximation replaces the root of the cut, *e.g.*, node 3, with a constant 0 that is then propagated to simplify the AIG. For example, if node 3 is replaced by a constant 0 in Figure 15a then the approximate AIG in Figure 15b is obtained. A SAT solver is then employed to compare the original AIG and the approximate AIGs to check wether the error constraint is violated, hence guaranteeing the error bound.

V. Approximate High-Level Synthesis

All of the aforementioned ALS techniques operate on either the gate-based netlist or on a Boolean representation



(a) Original AIG (b) Approximate AIG

Fig. 15: Approximating AIG-based representations. Dashed edges represent inverted signals. Approximate AIG is obtained by replacing node 3 in the original AIG with the constant 0 and simplifying the AIG accordingly.

of the circuit. In contrast, the goal of Approximate High-Level Synthesis (AHLS) is to integrate inexact operators as building blocks, in order to efficiently implement designs described in high-level languages such as behavioral Verilog or C language. Similarly to ALS techniques, input to AHLS flows is the original design description, which is analyzed and modified by the HLS tool to produce a candidate approximate design as outcome, as illustrated in Figure 16. The QoR of the candidate approximate design is evaluated with input test benches, with either simulation or analytical techniques. If the approximate design passes the QoR minimum requirement, then the design physical metrics, *e.g.*, power, timing and design area, are estimated to yield an approximate design variant. Since many approximate design candidates can be generated by the approximate HLS tool, a Pareto-optimal set of approximate designs is identified to provide the best trade-off between QoR and a physical metrics (*e.g.*, power).

One of the first techniques for HLS is ABACUS [38]. In ABACUS the behavioral or RTL Verilog description is first parsed to build its Abstract Syntax Tree (AST), as illustrated in Figure 17. A number of *transformation operators* are then applied to the AST to create approximate variants of the AST, which are then written back to Verilog and simulated for error evaluation, and synthesized for area and power evaluation. These operators include the following transformations.

1) *Bit-width simplifications:* This transformation reduces the bit width of signals by truncating them and removing some of the least significant bits. This transformation automatically leads to a reduction in the underlying logic resources that operate on these signals.
2) *Variable to constant substitutions:* Given the simulation results of the original circuit using the test benches, ABACUS analyzes the relative change in each signal, and if a signal has a relatively small range of values, then it is replaced by a constant that is equal to the mean value of the range. This substitution enables the logic synthesis tool to simplify the underlying logic.

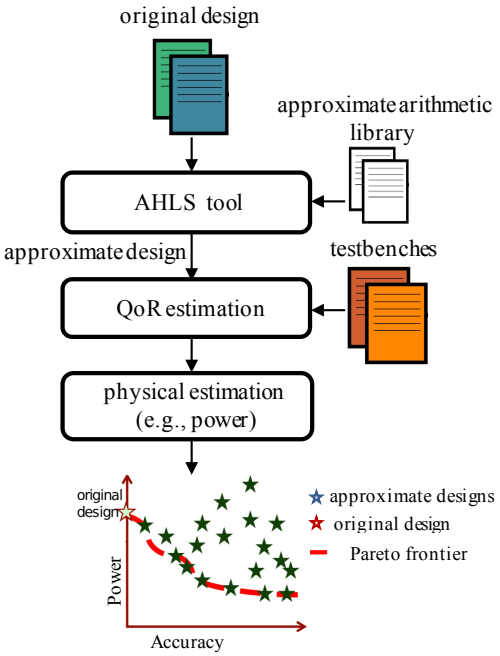


Fig. 16: General flow of AHLS methods.

3) *Approximate arithmetic transformations:* This transformation replaces an exact arithmetic operator (such as $*$ or $+$) by an approximate operator using the available approximate designs, such as DRUM [15], [16] and EvoApproxLib [35].
4) *Expression transformations:* For this transformation, ABACUS analyzes the arithmetic expressions in the design and replaces them by approximate versions; for example, a Verilog assignment statement such as `assign z = a*b+c*d` is replaced by `assign z = a*(b+d)`, where the value of signal `c` is approximated by the value of signal `a` and in the process saving one multiplier.
5) *Loop transformations:* In ABACUS, all Verilog loop are unrolled and the statements in the unrolled loop are then approximated using the aforementioned transformations

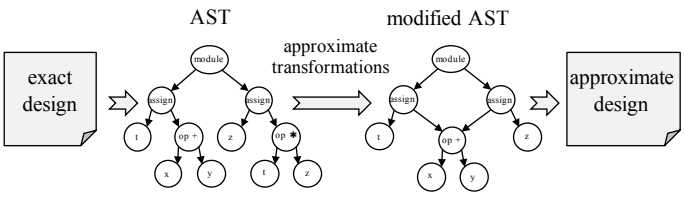


Fig. 17: ABACUS approximation flow. An input design is parsed and captured in Abstract Syntax Tree (AST) form. The AST is then manipulated to generate approximate ASTs that are written back to Verilog and synthesized into approximate circuits.

transformation operators in a random fashion and then selects and retains the approximate designs that have the optimal trade-off between accuracy and power consumption. This selection and optimization are achieved using an evolutionary approach based on the Non-dominated Sorting Genetic Algorithm (NSGA-II) [9]. ABACUS also prioritizes approximations on the critical path to create positive slack that can be exploited to reduce the voltage while keeping the frequency intact [37]. The reduction in voltage leads to additional power savings beyond those provided by the approximate logic.

Focusing on approximate arithmetic transformations, a few works (*e.g.*, [25], [34], [68]) propose a more detailed analysis for this transformation in order to choose a Pareto-optimal approximate operator from within a large library of approximate arithmetic operators [35]. For example, consider a statement such as `assign z=a+b`, where the exact adder is replaced by an approximate adder from a library of a large number of approximate adders offering optimal accuracy-energy trade-off for addition. To choose the best approximate adder, different approximate adders are substituted in the circuit and the impact of their errors is propagated to the circuit's outputs, and the final QoR and power consumption are evaluated. If the QoR meets a global error bound and the approximate designs offer a new Pareto point in terms of power-QoR trade-off, then the approximate design is accepted. To determine the best precisions, Constantinides *et al.* [6] formulate an integer linear program (ILP) to identify the best precision approximation for each arithmetic operator, such that the total energy is minimized subject to an upper bound on the output error variance. Their approach only considers bit-width simplifications as an avenue towards approximation. It is generalized by Li *et al.* [25], which also consider inexact implementations of adders and multipliers. In [25] the solution of the ILP formulation determines the best approximation for each arithmetic operator.

Raising the level of abstraction even further, Lee and Gerstlauer [24] propose an AHLS method that is capable of creating approximate designs from C-based descriptions of hardware systems. The main idea is to focus on applying approximations to loops since they are critical to the overall latency of digital systems. Rather than taking the ABACUS approach, where loops are unrolled and the resulting statements are approximated individually, the loop structure is kept intact. Instead, the iterations of a loop are split into a number of classes based on impact of each iteration on accuracy. For example, a loop with N iterations can be transformed into two: a *high-accuracy loop* with N_1 iterations, and a *low-accuracy loop* with N_2 iterations, such that $N = N_1 + N_2$. More aggressive approximations are then applied to the low-accuracy loop. For example, the low-accuracy loop can use low precision by using smaller bit widths for the variables, whereas the high-accuracy loop can use the fixed-point precision of the original loop. To analyze the iterations of a loop, the input C code is first compiled to an intermediate representation (IR) using LLVM [23]. Using test benches, the IR is then profiled to evaluate the data statistics, mobility, latency and energy of every operation in the design. The profiling considers approximate substitutions for each IR statement and

propagates the error to evaluate the QoR. This QoR estimation is done on a per iteration basis in order to classify each iteration based on its impact on accuracy. The iterations are then grouped together into classes based on their accuracy. For low-accuracy loops, the degree of approximation to the variables or arithmetic operations are chosen to minimize the latency or energy subject to QoR constraints.

Rolda et al. *et al.* [41] also propose a method for approximating loop nests, considering the case of linear equation solvers. They observe that, when using iterative methods, accuracy can be increased by either running more iterations or by performing each computation more precisely. They showcase that a combination of both strategies results in the best trade-offs between resources and accuracy.

VI. Comparative Evaluation of ALS Methods

As described in Sections III and IV, ALS methods proposed in literature have widely different characteristics. Most prominently, a number of works (described in Section III) introduce transformation strategies performed on graphs representing gate-level *netlists* of the circuit to be approximated, while others (covered in Section IV) perform the automatic rewriting of *Boolean functions* to be realized in hardware. In this section we conduct an empirical comparison of a number of available tools in the public domain, and provide a number of insightful results.

For the purpose of our experiments we construct a new open-source Benchmark set for Approximate Logic Synthesis (BACS).[3] Each benchmark is available in Verilog, and comes with a test bench, a QoR python script, and a sample approximate design. The characteristics of these benchmarks are given in Table I where we provide the total design area using the 45 nm FreePDK library [52] and using the Yosys-ABC synthesis framework [59]. We use these benchmarks in our experiments. The final quality of the synthesized and mapped netlists is a function of Yosys-ABC optimality.

In our first round of experiment, we focus on induced *Mean Absolute Errors* (MAEs) as a QoR metric to evaluate the impact of using ALS techniques when creating approximate arithmetic circuits. Area is instead employed as a cost metric. We consider an 8-bit unsigned multiplier and a 16-bit unsigned multiplier that are part of our BACS benchmark set. We apply three surveyed techniques to generate approximate variants for the multiplier: BLASYS [19] (which is a logic rewriting method),[4] EvoApprox [35] and GLP [45] (which, instead, perform structural netlist transformation). We also create baseline approximate multipliers (Truncated) by truncating the least significant bits of the operands before multiplication, as suggested in [3]. In Figure 18 and 19, we plot the relative area and mean absolute error (%) of the 8-bit and 16-bit multipliers in comparison to the accurate multipliers. We also plot the Pareto frontier for the designs with optimal trade-off between design area and MAE. The plot leads to the following insights:

- For the 8-bit multiplier, results show that BLASYS and EvoApprox produce comparable results with approximate designs from both techniques appearing on the Pareto frontier. The two techniques provide excellent approximate multipliers; for example, one implementation cuts the multiplier's area by half at the expense of only 0.32% mean absolute error (expressed as a percentage of the maximum circuit output).
- For the 16-bit multiplier, BLASYS shows better results as it is able to dominate the Pareto frontier. This better results show that BLASYS has better scability since it is able to partition the circuit for larger designs and then re-write the logic for each partition. GLP is dominated except for small values of MAE when it tends to produce reasonable results.
- BLASYS (and EvoApprox for 8-bit multiplier) dominate the manual approximate multipliers generated by truncating the least significant bits of the operands by a large margin. For instance, for an error threshold of 0.2%, EvoApprox offers a design with 45% area reduction compared to 18% area reduction for the truncated multiplier with the same error margin. This result shows that ALS techniques can produce better results than manual alternatives, while being labour-intensive and more flexible.

[3]BACS is available at https://github.com/scale-lab/BACS.
[4]BLASYS is available at https://github.com/scale-lab/BLASYS.

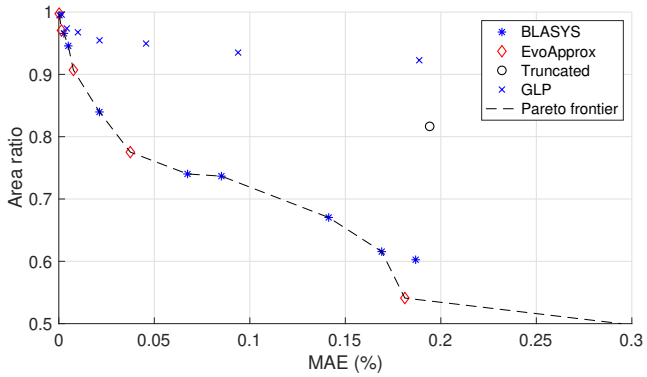


Fig. 18: Approximate design variants of an 8-bit unsigned multiplier. Approximate designs were created with BLASYS, EvoApprox, GLP and manually by truncating the least significant bits.

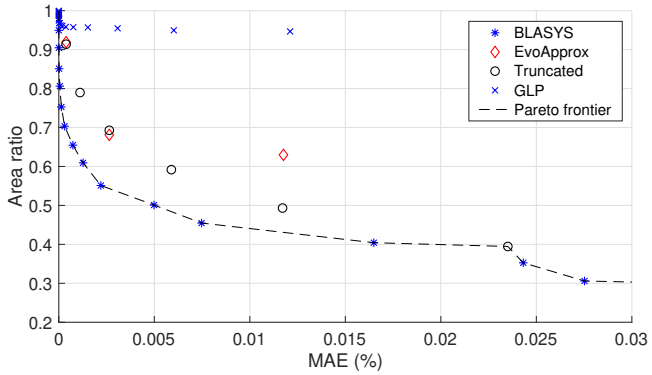


Fig. 19: Approximate design variants of a 16-bit unsigned multiplier. Approximate designs were created with BLASYS, EvoApprox, GLP and manually by truncating the least significant bits.

TABLE I: Characteristics of Benchmarks for Approximate Logic Synthesis (BACS).

Benchmark	I/O	Gates	Area (μm^2)	QoR Metric	Origin	desc.
abs_diff	16/9	70	194	MAE		absolute difference
adder32	64/33	176	512	MAE		32-bit adder
buttfly	32/34	168	464	MAE		simple butterfly structure
classifier	64/2	18788	56062	ER	[38]	SVM classifier
mac	12/8	101	268	MAE		combinational unit of a MAC circuit
fft	258/256	7184	23574	MSE	[38]	sequential 8-bit FFT circuit (sequential)
mult8	16/16	331	936	MAE		8-bit unsigned multiplier
mult16	32/32	1461	4214	MAE		16-bit unsigned multiplier
x2	10/7	29	65	HD	[12]	simple logic circuit

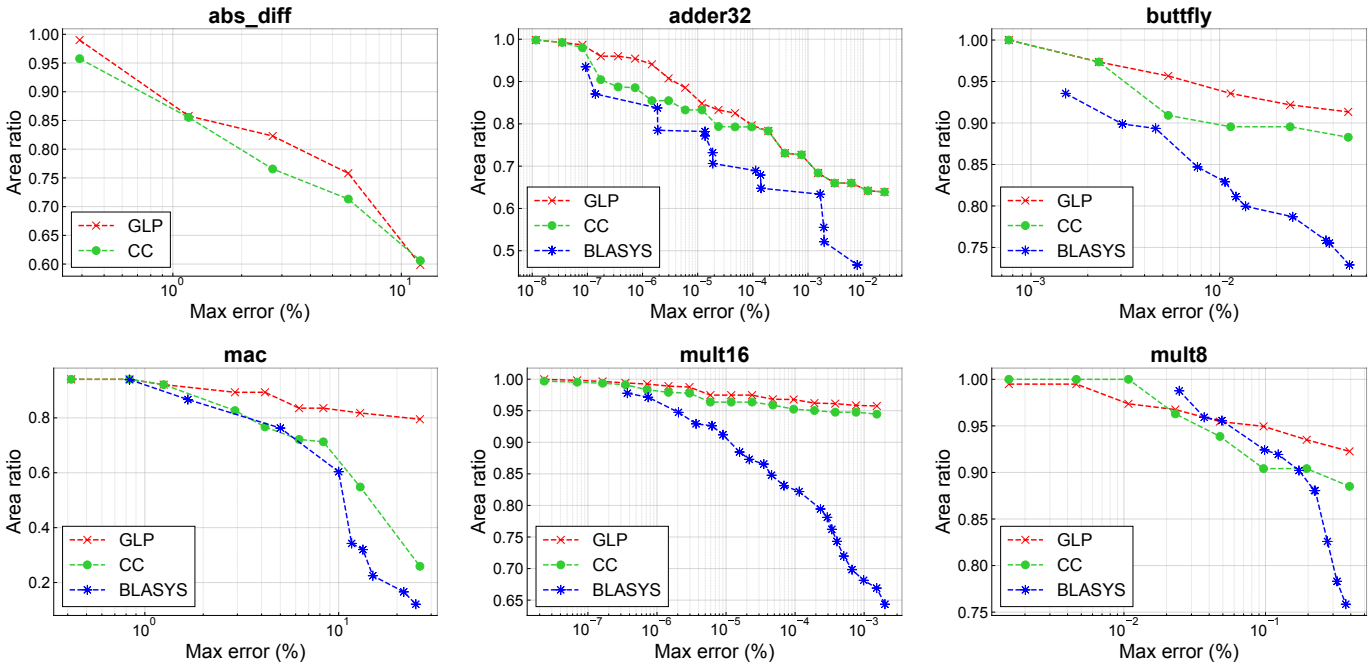


Fig. 20: Comparison of area reduction in approximate circuits for GLP [45], CC [44] and BLASYS [19]. The maximum absolute error (max error) is expressed as a percentage of the maximum circuit output.

A second comparison is provided in Figure 20, here the focus was set on containing the *worst case* error using ALS methods. We considered two structural transformation techniques, Circuit Carving (CC) [44] and Gate-Level Pruning (GLP) [45], and one technique based on logic rewriting (BLASYS [19]) over a subset of the BACS benchmark set. When set to control maximum error, ALS methods are necessarily more conservative than for average errors. However, only CC guarantees that the error at the output of an inexact circuit is indeed an upper threshold on the actual one, while the other two provide a soft bound based on Monte Carlo simulations on a subset of the circuit inputs, which they employ to assess the QoR at different approximation steps. Figure 20 reports the area ratio obtained by different constraints on such maximum error. All techniques provide good to excellent approximations, especially when considering the stricter error constraint. Their results are comparable for small error values, while for larger errors CC tends to dominate GLP, and BLASYS dominates both, except for the 8-bit absolute difference, which it was unable to process. For example, BLASYS halves the area with an error of 0.1% for the 32-bit adder (top row, central), while for the 4-bit multiply-add both CC and BLASYS reduce the original area to less than 30% with a 14% maximum error (bottom row, left). BLASYS again proves to be very effective in exploring the design space and scale to large circuits, while CC suffers from its exponential worst-case complexity, though it generally provides better approximations than GLP. Indeed, on larger error values, it has to stop before terminating exploration, and this leads to worse approximations. However, we note that in most real-world scenarios too large error values would not be meaningful, thus we can conclude that CC is a valid methodology when strict maximum error guarantees are required.

In our third comparison we contrast approximate logic rewriting methods with approximate HLS methods. We consider a combinational version of the 8-point FFT benchmark

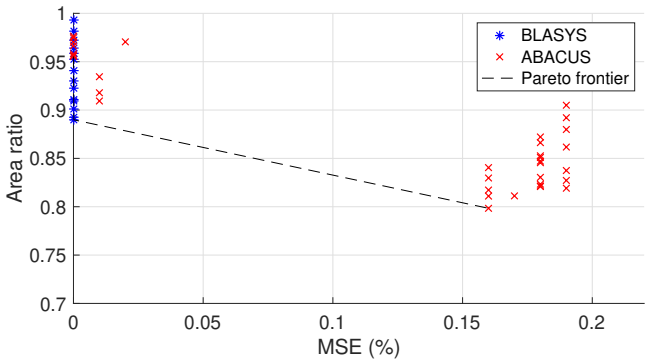


Fig. 21: Approximate design variants of an 8-point FFT circuit created using ABACUS and BLASYS.

written in Verilog RTL. We contrast ABACUS [38], which is an example of approximate HLS tools, against BLASYS [19] which relies on approximate logic rewriting.[5] For BLASYS, the circuit is first compiled to Boolean representation using Yosys. Figure 21 provides the error in MSE (%) versus area ratio compared to the accurate design for designs from both techniques. The results from ABACUS show two clusters of points with a big distance in between them with respect to MSE. For the first cluster with low MSE, BLASYS is able to achieve better results with smoother trade-off between accuracy and area as it incrementally approximates each partition in the design. However, this fine granularity comes at an expense: BLASYS is unable to make the drastic approximations required to reach the second cluster within reasonable runtime. For the provided results, the runtime of BLASYS was 22× the runtime of ABACUS. ABACUS is much faster than BLASYS because ABACUS performs its approximations on the abstract syntax tree, which has a size that is proportional to the RTL design text description. Thus, approximate HLS methods offers much better scalability in comparison to logic rewriting methods.

We summarize our insights from all experiments.

1) Structural transformation ALS methods require a synthesized circuit netlist as an input before any approximate transformation are applied. This requirement might in principle limit their effectiveness since the synthesized netlist can limit the set of reachable approximate netlists, if an incomplete set of transformation rules is used, as for example in gate-elimination-only methodologies. On the other hand, logic re-writing approaches do not assume a synthesized circuit, and the approximation is either conducted before or in conjunction with the synthesis process. Thus, logic re-writing can potentially lead to a larger design space exploration since synthesis is part of the process. However, due to the heuristic nature of the synthesis process, logic re-writing may, for large circuits, be unable to construct a good circuit structure. The overall conclusion must therefore be that for large circuits, structural approaches are beneficial when the structure of an exact circuit is a good approximation to the optimal structure of an approximation.
2) To achieve scalable results, it is important that approximate logic rewriting techniques either (1) rely on underlying scalable circuit data structures such as AIGs, or (2) use partitioning techniques to break-down the circuit into manageable subcircuits where logic re-writing can be applied on each subcircuit independently.
3) Automated ALS can provide approximate designs that dominate manually created ones, that exploit the knowledge of the underlying circuit structure.
4) Some ALS techniques can be adapted to a number of different error metrics (such as, for example, BLASYS) and others are instead built specifically for one given metric (such as for example CC, which is guaranteed to never exceed a maximum error, but cannot be easily adapted to average error).
5) When compared to approximate HLS, structural and rewriting methods often offer finer-grain control of the approximation because these methods operate at the gate or Boolean level. Thus, it is possible to create approximate circuits variants that have incremental nature with fine grain trade-off between QoR and design metrics such as area and power. Approximate HLS techniques operate at a higher level so their approximations tend to produce coarser results in terms of QoR and power trade-off. For the same reason, approximate HLS approaches are more scalable since they can produce in one approximation step what might take many steps in structural and re-writing ALS methods. An additional advantage for HLS methods over other methods is that their approximate designs are easier to interpret by human designers.

[5]ABACUS is available at https://github.com/scale-lab/ABACUS.

VII. Conclusions and Open Challenges in ALS

Approximate Logic Synthesis leads to resource-efficient hardware implementations that meet QoR requirements. Hence, it is extremely appealing in the current post-Moore era, where further exponential gains from advances in transistor technology cannot be taken for granted, and where applications are data-rich with high degree of error tolerance. Table II provides a taxonomy summary of the majority of papers reviewed in this survey. Despite the large improvements in state-of-the-art ALS in recent years, a number of hurdles still have to be confronted for ALS to become mainstream in hardware design, both regarding automation in low-level design of approximate components and methodologies for approximate high-level synthesis. We discuss the main challenges for ALS in the remainder of this section.

A. Synthesis of Approximate Circuits

Scalability. This aspect is still an issue for approximate design automation methodologies: as an example, ALS methods that rely on BDD and BMF for Boolean representation cannot handle large circuits, and as result need circuit partitioning methods to break down the circuit into manageable subcircuits [30]. However, this partitioning often reduces the optimality of results attained from these methods.

TABLE II: Summary of ALS works.

Reference	Synthesis Method	Error modeling / QoR evaluation	QoR Metric
Su *et al.* [53]	Venkataramani *et al.* [56]	Monte Carlo + change propagation matrix	error rate / average error
Zhang *et al.* [69]	Greedy gates pruning	signal values probability	approximate efficiency
Venkatesan *et al.* [58]	Equivalent untimed circuit	SAT / BDDs / Monte Carlo	error rate / average error / maximum error
Scarabottolo *et al.* [43]	Schlachter *et al.* [45]	partitioning + propagation matrices	maximum error
Liu *et al.* [27] & Vasicek *et al.* [54]	Netlist transformation: stochastic	Markov chain Monte Carlo	error rate / average error
Schlachter *et al.* [45]	Netlist pruning: greedy heuristics	Monte Carlo	error rate / average error
Venkataramani *et al.* [56]	Netlist manipulation: greedy heuristics	Monte Carlo	error rate / average error
Shin *et al.* [49]	Netlist pruning: greedy heuristics	Monte Carlo / Threshold testing	error rate / maximum error
Scarabottolo *et al.* [44]	Netlist pruning: exhaustive exploration	simulation	maximum error
Venkataramani *et al.* [31], [57]	Boolean rewriting: symbolic rewriting	symbolic	maximum error
Hashemi *et al.* [17], [19]	Boolean rewriting: BMF	Monte Carlo	average error
Soeken *et al.* [51], Frohilich *et al.* [13]	Boolean rewriting: BDDs	BDDs	maximum error
Chandrasekharan *et al.* [5]	Boolean rewriting: AIGs	SAT	maximum error
Nepal *et al.* [38]	High-level synthesis: Behavioral / RTL Verilog	Monte Carlo	SNR
Li *et al.* [25], Constantinides *et al.* [6]	High-level synthesis: precision exploration	analytic	error variance
Zervakis *et al.* [68], Mrazek *et al.* [34]	High-level synthesis: arithmetic exploration	simulation	average error
Lee *et al.* [24]	High-level synthesis: C	analytic	SNR
Rolda et al. *et al.* [41]	High-level Synthesis: C	simulation	average error / maximum error

Runtime accuracy-configurable hardware. An open research question regards the generation of designs having an approximate degrees which is not fixed at design time, but instead variable according to external (*e.g.*, battery level) and internal (*e.g.*, output confidence) factors. In such context, ALS could provide implementations for both the high-precision and high-efficiency operating modes, as well as the required logic to switch between either of the two. Nonetheless, while accuracy-configurable adders [47] and multipliers [1] have been proposed, their design is still manually done. Automated approaches, able to explore the benefits of accuracy configuration, in light of the overhead (energy- and area-wise) implied by the logic governing the transitions between different approximate settings, are instead currently lacking.

B. Approximate High Level Synthesis

Arithmetic Library. The synthesis of approximate operators is clearly a key first requirement for the wide adoption of ALS techniques. Except for very few works on adders [28] and multipliers [66], most of existing research on approximate arithmetic have focused on integer arithmetic disregarding floating point operators. Building robust AHLS tools require extensive libraries of approximate floating-point as well as fixed-point arithmetic components that need to be developed.

As covered in Section V, works in AHLS do study the composition of multiple approximate operators to realise approximate accelerators [25], [34], [38], [68]. Still, by picking the proper building blocks among the ones available in a library, such strategies cast accelerator design as a selection problem. Solutions are therefore restricted only to integrating the available library elements. More flexible approaches have indeed been proposed, but they only focus on signal width optimizations [8], [7], without considering the opportunities for logic simplification offered by ALS. These are instead leveraged by the strategies in [2] and [20], which employ interval arithmetic to estimate the impact, as seen at the output of a hardware module, of approximating its constituent operators. This stance allows the assessment of the operations approximability in advance of time-consuming synthesis and simulations. Nonetheless, these methodologies are only limited to combinatorial designs (a limited support for sequential ones is provided in [2]), and do not provide an explicit link between circuit-level QoR metrics (*e.g.*: average and maximum error) and application level ones (SNR, SSIM).

Cross-layer approximate design. Another little-explored aspect related to AHLS is that simplifications can be driven by algorithmic considerations [50] as well as ALS-based ones. Interaction between the two levels (*i.e.*, algorithmic and AHLS) has mainly been explored in an ad-hoc fashions, such as the design of linear equation solvers in [41] or that of classification engines in [11], while more general cross-layer approaches are still in their infancy [38].

Key towards the maturity of AHLS will be the development of strategies to quickly link error and QoR metrics across abstraction levels. On one side, such effort encompasses the development of tools dedicated to the early evaluation, from a high-level viewpoint, of approximation choices. Available frameworks are currently hampered by a narrow application scope, such has approximate neural networks [55]. On the other hand, methodologies should be envisioned that estimate the QoR impact of a simplifying transformation, the limiting or entirely avoiding post-hoc simulations. Such stance is far from trivial, since metrics of interest are usually related to a

considered scope: application-wide QoR being best expressed with metrics such as as classification accuracy or signal-to-noise ratio, while at the operator levels error rates or average errors are usually more appropriate.

REFERENCES

[1] O. Akbari, M. Kamal, A. Afzali-Kusha, and M. Pedram. Dual-quality 4: 2 compressors for utilizing in dynamic accuracy configurable multipliers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(4):1352–1361, Jan. 2017.

[2] G. Ansaloni, I. Scarabottolo, and L. Pozzi. Judiciously spreading approximation among arithmetic components with top-down inexact hardware design. In *Applied Reconfigurable Computing*, pages 14–29. Springer International Publishing, Mar. 2020.

[3] B. Barrois, O. Sentieys, and D. Menard. The hidden cost of functional approximation against careful data sizing—a case study. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 181–186, Mar. 2017.

[4] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.

[5] A. Chandrasekharan, M. Soeken, D. Große, and R. Drechsler. Approximation-aware rewriting of aigs for error tolerant applications. In *Proceedings of the International Conference on Computer Aided Design*, page 83, Nov. 2016.

[6] G. Constantinides, P. Cheung, and W. Luk. Optimum wordlength allocation. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 219–228, 2002.

[7] G. Constantinides, A. Kinsman, and N. Nicolici. Numerical data representations for FPGA-based scientific computing. *IEEE Design and Test of Computers*, 28(4):8–17, May 2011.

[8] G. A. Constantinides, P. Cheung, and W. Luk. The multiple wordlength paradigm. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, pages 51–60, 2001.

[9] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II. In *International Conference on Parallel Problem Solving from Nature*, pages 849–858, 2000.

[10] T. Drane, T. Rose, and G. A. Constantinides. On the systematic creation of faithfully rounded truncated multipliers and arrays. *IEEE Transactions on Computers*, 63(10):2513–2525, October 2014.

[11] L. Ferretti, G. Ansaloni, L. Pozzi, A. Aminifar, D. Atienza, L. Cammoun, and P. Ryvlin. Tailoring SVM inference for resource-efficient ECG-based epilepsy monitors. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1–4, Mar. 2019.

[12] P. Fišer and J. Schmidt. A comprehensive set of logic synthesis and optimization examples. In *Proc. of 12th Int. Workshop on Boolean Problems (IWSBP)*, pages 151–158, 2016.

[13] S. Fröhlich, D. Große, and R. Drechsler. Error Bounded Exact BDD Minimization in Approximate Computing. In *International Symposium on Multi-Level Logic*, pages 254–259, 2017.

[14] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *IEEE European Test Symposium (ETS)*, pages 1–6, May 2013.

[15] S. Hashemi, R. I. Bahar, and S. Reda. DRUM: A Dynamic Range Unbiased Multiplier for Approximate Applications. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 418–425, 2015.

[16] S. Hashemi, R. I. Bahar, and S. Reda. A low-power dynamic divider for approximate applications. In *IEEE/ACM Design Automation Conference*, number 105, 2016.

[17] S. Hashemi and S. Reda. Generalized Matrix Factorization Techniques for Approximate Logic Synthesis. In *ACM/IEEE Design Automation and Test in Europe*, pages 1289–1292, 2019.

[18] S. Hashemi, H. Tann, F. Buttafuoco, and S. Reda. Approximate computing for biometric security systems: A case study on iris scanning. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 319–324, March 2018.

[19] S. Hashemi, H. Tann, and S. Reda. BLASYS: Approximate logic synthesis using boolean matrix factorization. In *Proceedings of the 55th Design Automation Conference*, pages 55:1–55:6, June 2018.

[20] J. Huang, J. Lach, and G. Robins. A methodology for energy-quality tradeoff using imprecise hardware. In *Proceedings of the 49th Design Automation Conference*, pages 504–509. IEEE, June 2012.

[21] A. B. Kahng and S. Kang. Accuracy-configurable adder for approximate arithmetic designs. In *DAC Design Automation Conference 2012*, pages 820–825, 2012.

[22] P. Kulkarni, P. Gupta, and M. Ercegovac. Trading accuracy for power with an underdesigned multiplier architecture. In *Proceedings of the 24th International Conference on VLSI Design*, pages 346–351, Jan. 2011.

[23] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *ACM roceedings of the international symposium on Code generation and optimization:*, pages 75–84, 2004.

[24] S. Lee, L. K. John, and A. Gerstlauer. High-level synthesis of approximate hardware under joint precision and voltage scaling. In *IEEE/ACM Design Automation Conference*, pages 187–192, 2017.

[25] C. Li, W. Luo, S. S. Sapatnekar, and J. Hu. Joint precision optimization and high level synthesis for approximate computing. In *IEEE/ACM Design Automation Conference*, number 104, pages 1–6, 2015.

[26] A. Lingamneni, C. Enz, K. Palem, and C. Piguet. Synthesizing parsimonious inexact circuits through probabilistic design techniques. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):93, May 2013.

[27] G. Liu and Z. Zhang. Statistically certified approximate logic synthesis. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 344–351, Nov. 2017.

[28] W. Liu, L. Chen, C. Wang, M. O?Neill, and F. Lombardi. Design and analysis of inexact floating-point adders. *IEEE Transactions on Computers*, 65(1):308–314, Mar. 2015.

[29] W. Liu, F. Lombardi, and M. Shulte. A retrospective and prospective view of approximate computing [point of view. *Proceedings of the IEEE*, 108(3):394–399, 2020.

[30] J. Ma, S. Hashemi, and S. Reda. Approximate Logic Synthesis Using BLASYS. In *Workshop on Open-Source EDA Technology*, number 5, 2019.

[31] J. Miao, A. Gerstlauer, and M. Orshansky. Approximate logic synthesis under general error magnitude and frequency constraints. In *Proceedings of the International Conference on Computer Aided Design*, pages 779–786, Nov. 2013.

[32] P. Miettinen and J. Vreeken. Model order selection for boolean matrix factorization. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 51–59, 2011.

[33] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-aware AIG rewriting a fresh look at combinational logic synthesis. In *IEEE/ACM Design Automation Confereence*, pages 532–535, 2006.

[34] V. Mrazek, M. A. Hanif, Z. Vasicek, L. Sekanina, and M. Shafique. autoAx: An Automatic Design Space Exploration and Circuit Building Methodology utilizing Libraries of Approximate Components. In *IEEE/ACM Design Automation Conference*, volume 123, pages 1–6, 2019.

[35] V. Mrazek, Z. Vasicek, and L. Sekanina. Evoapproxlib: Extended library of approximate arithmetic circuits. In *Workshop on Open-Source EDA Technology*, number 10, pages 1–4, 2019.

[36] K. E. Murray, A. Suardi, V. Betz, and G. Constantinides. Calculated Risks: Quantifying Timing Error Probability with Extended Static Timing Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(4):719–732, Mar. 2018.

[37] K. Nepal, S. Hashemi, H. Tann, R. I. Bahar, and S. Reda. Automated High-Level Generation of Low-Power Approximate Computing Circuits. *IEEE Trans. Emerging Topics Computing*, 7(1):18–30, 2016.

[38] K. Nepal, Y. Li, R. I. Bahar, and S. Reda. ABACUS: A Technique for Automated Behavioral Synthesis of Approximate Computing Circuits. In *IEEE/ACM Design, Automation and Test in Europe*, pages 1–6, 2014.

[39] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan. Aslan: Synthesis of approximate sequential circuits. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1–6, Mar. 2014.

[40] S. Rehman, W. El-Harouni, M. Shafique, A. Kumar, and J. Henkel. Architectural-space exploration of approximate multipliers. In *Proceedings of the International Conference on Computer Aided Design*, pages 1–8, Nov. 2016.

[41] A. Roldao-Lopes, A. Shahzad, G. A. Constantinides, and E. C. Kerrigan. More flops or more precision? accuracy parameterizable linear equation solvers for model predictive control. In *Proceedings of the 17th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 209–216. IEEE, Apr. 2009.

[42] H. Saadat, H. Javaid, and S. Parameswaran. Approximate integer and floating-point dividers with near-zero error bias. In *Proceedings of the*

56th Annual Design Automation Conference 2019, pages 161:1–161:6. ACM, June 2019.
[43] I. Scarabottolo, G. Ansaloni, G. Constantinides, and L. Pozzi. Partition and Propagate: an error derivation algorithm for the design of approximate circuits. In *Proceedings of the 56th Design Automation Conference*, pages 1–6, June 2019.
[44] I. Scarabottolo, G. Ansaloni, and L. Pozzi. Circuit Carving: A methodology for the design of approximate hardware. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 545–550, Mar. 2018.
[45] J. Schlachter, V. Camus, K. V. Palem, and C. Enz. Design and applications of approximate circuits by gate-level pruning. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(5):1694–1702, Feb. 2017.
[46] E. Sentovich and K. Singh. A system for sequential circuit synthesis. Technical report, EECS, UCB, 1992.
[47] M. Shafique, W. Ahmad, R. Hafiz, and J. Henkel. A low latency generic accuracy configurable adder. In *Proceedings of the 52nd Design Automation Conference*, pages 1–6, July 2015.
[48] K. Shi, D. Boland, E. Stott, S. Bayliss, and G. A. Constantinides. Datapath synthesis for overclocking: Online arithmetic for latency-accuracy trade-offs. In *Proceedings of the Design Automation Conference 2014*. ACM, 2014.
[49] D. Shin and S. K. Gupta. A new circuit simplification method for error tolerant applications. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1–6, Mar. 2011.
[50] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *SIGSOFT symposium and European conference on Foundations of software engineering*, pages 124–134, Sept. 2011.
[51] M. Soeken, D. Große, A. Chandrasekharan, and R. Drechsler. BDD minimization for approximate computing. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 474–479, Jan. 2016.
[52] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, et al. FreePDK: An open-source variation-aware design kit. In *2007 IEEE international conference on Microelectronic Systems Education (MSE'07)*, pages 173–174, June 2007.
[53] S. Su, Y. Wu, and W. Qian. Efficient batch statistical error estimation for iterative multi-level approximate logic synthesis. In *Proceedings of the 55th Design Automation Conference*, pages 54:1–54:6, June 2018.
[54] Z. Vasicek and L. Sekanina. Evolutionary approach to approximate digital circuits design. *IEEE Transactions on Evolutionary Computation*, 19(3):432–444, July 2015.
[55] F. Vaverka, V. Mrazek, Z. Vasicek, and L. Sekanina. TFApprox: Towards a Fast Emulation of DNN Approximate Hardware Accelerators on GPU. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Mar. 2020.
[56] S. Venkataramani, K. Roy, and A. Raghunathan. Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1367–1372, Mar. 2013.
[57] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan. SALSA: systematic logic synthesis of approximate circuits. In *Proceedings of the 49th Design Automation Conference*, pages 796–801, June 2012.
[58] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan. MACACO: Modeling and analysis of circuits for approximate computing. In *Proceedings of the International Conference on Computer Aided Design*, pages 667–673, Nov. 2011.
[59] C. Wolf. Yosys open synthesis suite, 2016.
[60] Y. Wu and W. Qian. An efficient method for multi-level approximate logic synthesis under error rate constraint. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2016.
[61] Q. Xu, T. Mytkowicz, and N. Kim. Approximate computing: A survey. *IEEE Design and Test*, 33(1):8–22, Jan. 2016.
[62] C. Yang, M. Ciesielski, and V. Singhal. Bds: A bdd-based logic optimization system. In *ACM/IEEE Design Automation Conference*, pages 866–876, 2000.
[63] S. Yang. Logic synthesis and optimization benchmarks user guide. In *Technical Report Microelectronics Center of North Carolina,*, 1991.
[64] Y. Yao, S. Huang, C. Wang, Y. Wu, and W. Qian. Approximate disjoint bi-decomposition and its application to approximate logic synthesis. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 517–524, Nov. 2017.
[65] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu. On reconfiguration-oriented approximate adder design and its application. In *Proceedings of the International Conference on Computer Aided Design*, pages 48–54, Nov. 2013.
[66] P. Yin, C. Wang, W. Liu, and F. Lombardi. Design and performance evaluation of approximate floating-point multipliers. In *IEEE Computer Society Annual Symposium on VLSI*, pages 296–301, July 2016.
[67] A. C. B. Z. Wang, H. R. Sheikh, and E. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.
[68] G. Zervakis, S. Xydis, D. Soudris, and K. Pekmestzi. Multi-Level Approximate Accelerator Synthesis Under Voltage Island Constraints. *IEEE Transactions on Circuits and Systems Part II: Express Briefs*, 66(4):607–611, 2019.
[69] Z. Zhang, Y. He, J. He, X. Yi, Q. Li, and B. Zhang. Optimal slope ranking: An approximate computing approach for circuit pruning. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, May 2018.
[70] N. Zhu, W. L. Goh, W. Zhang, K. S. Yeo, and Z. H. Kong. Design of low-power high-speed truncation-error-tolerant adder and its application in digital signal processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(8):1225–1229, 2010.

Ilaria Scarabottolo received her B.Sc. degree in Mathematical Engineering and her M.Sc. degree in Computer Engineering at Politecnico di Milano, Italy, in 2013 and 2016. In parallel, she has obtained a second M.Sc degree at École Centrale Paris, France, in 2014. Since 2016, she has been a PhD student at the Università della Svizzera Italiana (USI), Lugano, Switzerland, Faculty of Informatics. In 2019, she was awarded the Swiss National Foundation grant DOC-Mobility, thanks to which she spent six months as visiting student at Imperial College London. Her research interests include approximate logic synthesis for error-tolerant applications, embedded systems and low-power hardware design.

Giovanni Ansaloni Giovanni Ansaloni is currently a post-doctoral researcher at the Faculty of Informatics of Università della Svizzera Italiana (USI-Lugano, Switzerland). From 2011 to 2015, he was a researcher at EPFL (Lausanne, Switzerland). He received the M.Sc. degree in Electronic Engineering from University of Ferrara (Italy) in 2003, the MAS degree from the ALaRI institute (Switzerland) in 2005 and the Ph.D. Degree from USI-Lugano in 2011. His research efforts cover different areas related to hardware/software co-design, including high-level synthesis, domain-specific architectures, and low-power signal processing.

George Constantinides received the PhD degree from Imperial College London in 2001. Since 2002, he has been with the faculty at Imperial College London, where he is currently Professor of Digital Computation and Head of the Circuits and Systems research group. He was General Chair of the ACM/SIGDA International Symposium on Field-programmable Gate Arrays in 2015. He serves on several program committees and has published over 200 research papers in peer-refereed journals and international conferences. Prof. Constantinides is a Senior Member of the IEEE and a Fellow of the British Computer Society.

Laura Pozzi received the Ph.D. degree in computer engineering from Politecnico di Milano, Milan, Italy, in 2000. She is currently a Professor with the Faculty of Informatics, Università della Svizzera Italiana, Lugano, Switzerland. She was a Post-Doctoral Researcher with EPFL, Lausanne, Switzerland; a Research Engineer with STMicroelectronics, San Diego; and an Industrial Visitor with University of California at Berkeley. Her current research interests include automating embedded processor customization, high performance compiler techniques, innovative reconfigurable fabrics, high-level synthesis design space exploration, and approximate computing. Prof. Pozzi has served as an Associate Editor for the IEEE Transactions on Computer-Aided Design and the IEEE Design and Test, and is or has been in the Technical Program Committee of several international conferences in the areas of compilers and architectures for embedded systems.

Sherief Reda Sherief Reda received the Ph.D. degree in Computer Science and Engineering from the University of California, San Diego in 2006. He is currently a Full Professor with the School of Engineering, Brown University, Providence, RI. His research interests include energy-efficient computing, thermal-power sensing and management, low-power design techniques, and design automation. He has over 120 publications in peer-reviewed conferences and journals with several of them receiving best paper nominations and awards. He serves as an associate editor for IEEE Transactions on Computer-Aided Design. He is a senior member of IEEE.