INVITED PAPER

Approximate Logic Synthesis: A Survey

By 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 the 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 guides 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 the 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.

KEYWORDS | Approximation; circuit; logic synthesis.

I. INTRODUCTION

Given an intended functionality, established methodologies for hardware design focus on achieving good tradeoffs between performance metrics (e.g., latency and 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 the input 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 of the tolerated implementation inaccuracy, as shown in Fig. 1(b).

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. Hence, ALS is 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 Fig. 1(a). 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 Han and Orshansky [14], Liu *et al.* [29], and Xu *et al.* [61]. Approximate circuit synthesis is particularly attractive since approximate circuits are employed as basic blocks for realizing application-specific accelerators that are highly relevant components of modern system-on-chips [18].

Approximate circuit synthesis has the ability to automate the process of discovering approximate implementations, given an exact circuit description. For example, in Fig. 2, we provide an illustration of the ability of approximate synthesis techniques to generate a large number of approximate variants of an eight-point fast Fourier transform (FFT) circuit, where each point reports the

Manuscript received January 22, 2020; revised May 11, 2020; accepted July 19, 2020. The work of Ilaria Scarabottolo, Giovanni Ansaloni, and Laura Pozzi was supported in part by the ADApprox (Grant: 200020-188613) and ML-edge (Grant: 200020_182009) projects funded by the Swiss NSF and in part by the MyPreHealth (Grant: 16073) project funded by Hasler Stiftung. The work of George A. Constantinides was supported by the EPSRC project EP/P010040/1, the EPSRC project EP/K034448/1, Imagination Technologies, and the Royal Academy of Engineering. The work of Sherief Reda was supported by NSF under Grant 1814920 and DoD ARO under Grant W911NF-19-1-0484. *(Corresponding author: Ilaria Scarabottolo.)*

Ilaria Scarabottolo, **Giovanni Ansaloni**, and **Laura Pozzi** are with the Faculty of Informatics, Università della Svizzera Italiana (USI), 6900 Lugano, Switzerland (e-mail: ilaria.scarabottolo@usi.ch).

George A. Constantinides is with the Department of Electrical and Electronic Engineering, Imperial College London, London SW7 2AZ, U.K.

Sherief Reda is with the School of Engineering, Brown University, Providence, RI 02912 USA (e-mail: sherief_reda@brown.edu).

Digital Object Identifier 10.1109/JPROC.2020.3014430

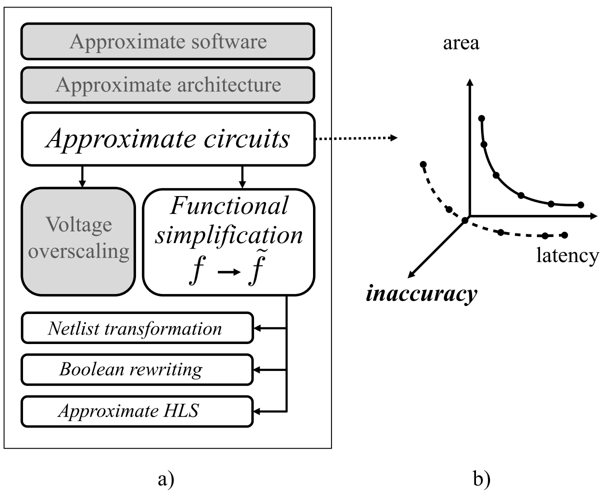


Fig. 1. ***(a) AC techniques taxonomy where areas discussed in this article are highlighted. (b) Inaccuracy represents a new dimension for circuit synthesis.***

results of one approximate design. For Fig. 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 Fig. 2 show the potential of automated ALS: 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 a part of the benchmarks for approximate circuit synthesis (BACS) that we release in conjunction with this article.[1]

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, that is, 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 tradeoff but again limit the analysis to adders and multipliers only. Unlike these specific handcrafted 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 in Fig. 1(a): 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, as shown in Fig. 3: netlist transformation, Boolean rewriting, and approximate high-level synthesis (AHLS).

In netlist transformation [see Fig. 3(a)], the Boolean function f is already mapped into a netlist, that is, 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 of 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 shown in Fig. 3(b): on the right column, some values in bold have been flipped with respect to the original truth table.

Finally, AHLS focuses on the highest level of abstraction for ALS, where the function is described at behavioral levels, such as in RTL Verilog or C language. An example

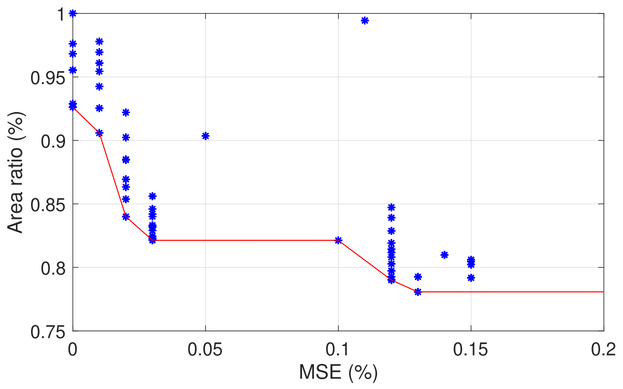
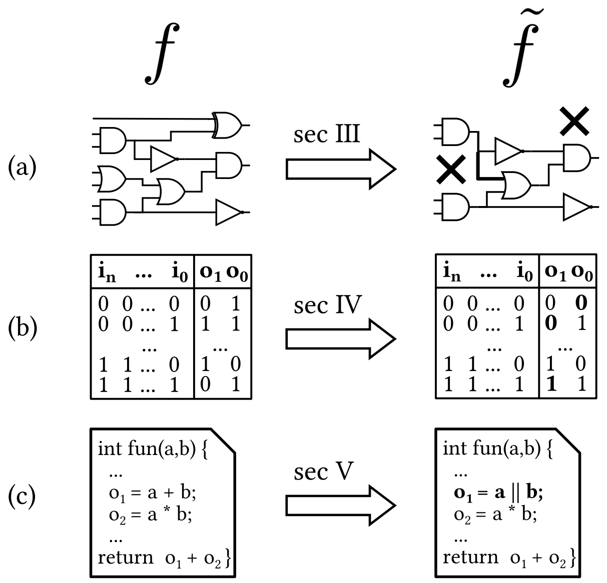


Fig. 2. ***Approximate design variants of an eight-point FFT circuit generated using the ABACUS tool. Each point represents the error and design area of an approximate circuit compared with the original design. MSE stands for mean square error.***

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



i_n	...	i_0	o_1	o_0
0 0	...	0	0	1
0 0	...	1	1	1
	...			...
1 1	...	0	1	0
1 1	...	1	0	1

i_n	...	i_0	o_1	o_0
0 0	...	0	0	**0**
0 0	...	1	**0**	1
	...			...
1 1	...	0	1	0
1 1	...	1	**1**	1

```
int fun(a,b) {
  ...
  o1 = a + b;
  o2 = a * b;
  ...
return  o1 + o2}
```

```
int fun(a,b) {
  ...
  o1 = a || b;
  o2 = a * b;
  ...
return  o1 + o2}
```

Fig. 3. ***Possible functional simplification approaches are illustrated: a generic Boolean function f is transformed in f̃, either acting on a synthesized netlist, for instance, at the gate level (a), or on its truth table (b). Finally, the circuit at behavioral level can be simplified by AHLS (c).***

fragment of such code is depicted in Fig. 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 AHLS are surveyed in Section V.

Regardless of the ALS technique employed for simplification, the result is an approximate 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 is dedicated to error models and quantification for approximate circuits, since precise error estimation is a cornerstone in AC. To appreciate the different phases in which error estimation is needed, Fig. 4 illustrates a typical ALS 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 Fig. 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 Fig. 4—to verify whether output quality constraints are satisfied in the synthesized approximate circuit.

Note that not all ALS methods reviewed in Sections III–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.

This article is organized 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–V describe the strategies for approximate hardware synthesis depicted in Fig. 3. In particular, Section V discusses the 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, the 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 Fig. 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 and Sekanina [54] employed a genetic programming technique that does not need the guide of *a priori* error modelingsince it automatically evolves toward lower error circuits, while other works, such as [44], compute a tight bound on the maximum error in the first phase, and then, the resulting simplified circuits do not need to be reevaluated 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 methods used for error modeling.

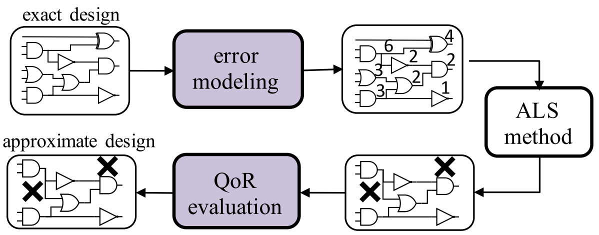


Fig. 4. ***Phase of error modeling can precede ALS, with the aim of decorating a circuit (or a Boolean specification) with a notion of error, hence guiding subsequent logic-simplification decisions. Then, a phase of QoR evaluation occurs to verify whether output quality constraints are satisfied in the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

A. Error Metrics

When performing error profiling, the first step is to appropriately encode the bits at the output according to the intended representation (e.g., as signed or unsigned numbers). Hence, the difference d between an exact and approximate implemented Boolean functions (f and $\tilde{f}$, respectively) can be computed between the 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, and so on. For example, Ma *et al.* [30] and Venkataramani *et al.* [57] employed the Hamming distance (HD) as a measure of d, defined as the number of bit flips in $\tilde{f}$ with respect to the original f.

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

When, as done in [5] and [43]–[45], the focus is on controlling the maximum error (i.e., worst case distance), which 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 $\tilde{f}$, where X is the set of all possible circuit inputs and x is a generic input.

Several ALS 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 above expression 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, normalized 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 (ER) is a common metric that captures this phenomenon. For a generic circuit, given $W = \{x \in X | f(x) \neq \tilde{f}(x)\}$, that is, the set of inputs for which the approximate function computes an erroneous output, the ER is defined as

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

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

Some works introduce further metrics for error estimation that also takes into account the structure of the circuit being simplified, in addition to its functionality. Notably, Zhang *et al.* [69] adopted the notion of approximate efficiency, which is 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: $\Delta\text{EDP}/D$. 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, which is the 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. The signal-to-noise ratio (SNR) is evaluated for signal-processing designs [24], [25], while structural similarity (SSIM [67]) is used in image-processing ones [34]. Furthermore, classification accuracy assesses QoR in [20] and [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, precise computation of the error caused by a circuit modification requires an 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]

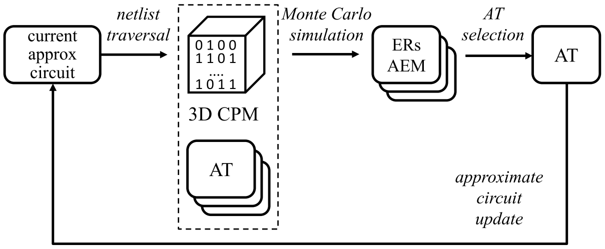


Fig. 5. ***Block scheme of the batch error estimation technique in [53], where, at each iteration, a set of candidate ATs is evaluated through the CPM to obtain the corresponding ER and AEM.***

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 ERs) and error bounds (for maximum errors). In the following, we review the body of work for such methods.

1) 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 (ATs).

Su *et al.* [53] devise a technique that effectively lowers the computational effort entailed. Their strategy, as shown in Fig. 5, aims at estimating the error introduced by a set of candidate ATs without having to resort to the Monte Carlo simulation for each of them. They propose a two-step method that computes a 3-D 0-1 change propagation matrix (CPM) and 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 is the number of nodes in the netlist being simplified, and O is the cardinality of its output. An entry in CPM $[i, n, o]$ is 1 if and only if a change in the node n propagates to the output o under the ith pattern. The CPM is computed in a reverse topological traverse of the netlist, starting from the primary outputs and then recursively calculating the entries for each node fan-in.

The same CPM can then be employed to evaluate, within a single Monte Carlo pass, the ER, and the average error magnitude (AEM) derived from all candidate transformations. The batch calculation simply computes the desired error metric for each AT individually, but, because of the CPM, there is no need to rerun Monte Carlo simulation at each step since information on the error propagation toward 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/ER constraint is violated. The computational complexity of this strategy is $\mathcal{O}(MOT)$, where T is the size of the set of possible ATs, compared with $\mathcal{O}(MNT)$ 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.

2) Bounding Maximum Errors: Monte Carlo-based approaches are not employable when maximum error thresholds must be provided, as they cannot account for outliers. To compute error guarantees, Schlachter *et al.* [45] introduced an algorithm that 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, that is, 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. Hence, the weight of a node i is

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

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

As shown in Fig. 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 into 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 subgraph truth tables, deriving their propagation matrices M. The relation between a subgraph input weights' vector $\mathbf{w}_{\text{in}}$ and output weights' vector $\mathbf{w}_{\text{out}}$ is then

$$\mathbf{w}_{\text{in}} = M\mathbf{w}_{\text{out}}.$$

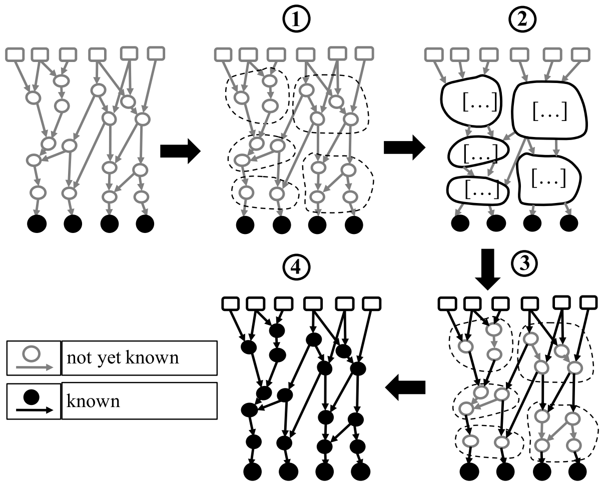


Fig. 6. ***Four steps in the P&P methodology from [43]. 1: partitioning. 2: derivation of the propagation matrices. 3: computation of the weights across partitions. 4: subgraph simulations.***

3) *Propagation:* Weights are then propagated across subgraphs considering them in the reverse topological order. If the children nodes of a subgraph output belong to different subgraphs, the subgraph output node weight $\mathbf{w}_{\text{out}}$ is conservatively set as the sum of the $\mathbf{w}_{\text{in}}$ 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 is the number of subgraphs, and E is the number of edges traversing distinct subgraphs.

III. ALS: STRUCTURAL NETLIST TRANSFORMATIONS

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

1) greedy heuristics for netlist pruning;
2) greedy heuristics for netlist manipulation;
3) stochastic netlist transformation;
4) exhaustive exploration for netlist pruning.

A. Greedy Heuristics for Netlist Pruning

Shin and Gupta [49] employed 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 toward the primary inputs, marks all nodes whose fan-out is now empty as deletable, and eliminates marked nodes.
2) *Forward simplification:* It performs the same operation in the forward direction, traversing the SAF fan-outs toward 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.

GLP by Schlachter *et al.* [45] presents another greedy iterative algorithm for circuit simplification. The proposed framework, as shown in Fig. 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 that 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 the significance is computed in a reverse topological graph traversal, as mentioned in Section II-B, starting from the primary outputs' arithmetic bit-significance and then assigning to each node i the significance σ_i

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

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

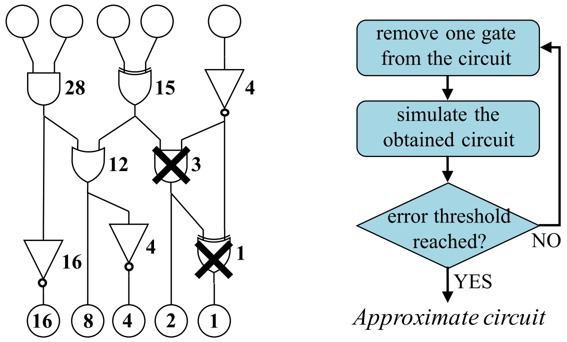


Fig. 7. ***GLP [45] framework. Gates are removed from a generic circuit starting from the least significant, until the allowed error threshold is reached.***

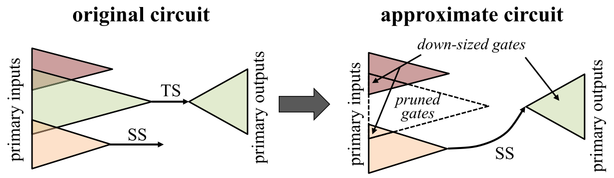


Fig. 8. *SASIMI [56] illustration of signal substitution: when TS is replaced by SS, gates belonging exclusively to TS fan-in cone are deleted, while those shared with other signals are downsized.*

resynthesizes the circuit, simulates it with a Monte Carlo process to verify that the error constraints on ER 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 (15–20 min for a 32-bit adder with five million input combinations). However, it allows the selection between a wide 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] proposed another greedy strategy called Substitute-And-SIMplIfy (SASIMI). In SASIMI, the 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 article is illustrated in Fig. 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 fan-out can potentially be reduced, as well as that belonging to TS fan-in and to other signals' fan-in. Therefore, both direct pruning and indirect downsizing are considered in the choice of TS.

It is clear that 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 $P_{\text{DIFF}}(1 - P_{\text{DIFF}})$.

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 ER and average absolute error magnitude on a subset of all possible circuit inputs that are assumed to be uniformly distributed. The algorithm takes as input the original circuit and a target error and 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 and Zhang [27] argue that the assumption of uniform distribution of input data is seldom correct. Therefore, they propose Statistically Certified ALS (SCALS), an iterative framework that presents some radical differences with respect to those described in Sections III-A and III-B. First, the authors of SCALS apply statistical hypothesis testing to estimate the errors obtained on the circuit outputs after an AT to guarantee that the population behavior is, indeed, a faithful representation of the actual data distribution.

Second, their algorithm presents two nested iterative loops, as shown in Fig. 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 subnetlists, which will be independently optimized, in parallel, through the inner loop in Fig. 9 (right).

The resulting optimized subnetlists are then recombined, and the resulting netlist error is evaluated through statistical hypothesis testing. Such a 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

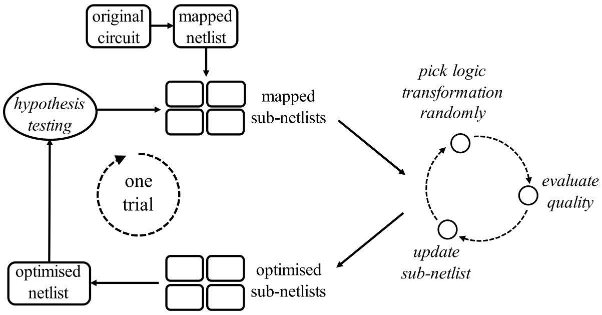


Fig. 9. *Nested-loops approach of SCALS [27]. For each trial, all mapped subnetlists are optimized in parallel through iterative random selection of possible transformations. Once the optimized subnetlists are recomposed, the error of the circuit is assessed through hypothesis testing.*

error metrics, namely ER and average relative error magnitude.

It is interesting to understand how the subnetlists optimization (the inner loop) works: SCALS considers a set of logic transformations $T = E \cup A$, where $E = \{\text{BALANCE, REWRITE, REFACTOR}\}$ represents the exact transformations, and $A = \{\text{REDUCE, FLIP, ADD}\}$ represents the 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, ATs do alter it. REDUCE and FLIP randomly pick a logic gate from the netlist: the first removes one of its fan-ins, 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.

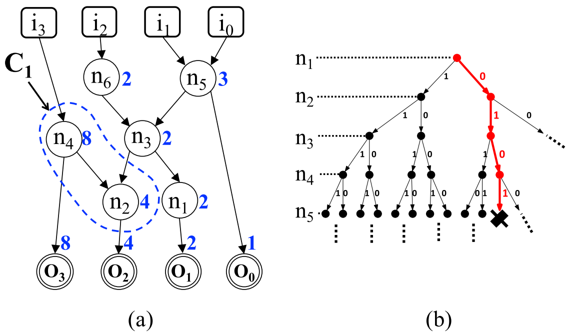
For each logic transformation, the approximate subnetlist is mapped to the desired technology to assess its quality. Statistical inference techniques are used to estimate the errors introduced in this phase since hypothesis testing for each transformation would be computationally infeasible. If the quality improves, the subnetlists 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 [54] proposed EvoApprox, a genetic algorithm to mutate the circuit into approximate versions by swapping gates with wire connections. 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 2-D grid, that is, 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 the 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.

1) *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.
2) *Error-oriented method:* The area is minimized while keeping the error between a user-specified range.
3) *Multiobjective:* 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.



(a) (b)

Fig. 10. ***(a) Example of cut for CC [44], with significance given by its output nodes. (b) Corresponding path in the binary tree search, where the 1-branch is taken for node n_2 and n_4 only.***

D. Exhaustive Exploration for Netlist Pruning

As opposed to other works surveyed in this section, circuit carving (CC) by Scarabottolo *et al.* [44] does not employ iterative approximations toward inexact logic synthesis. It, instead, resorts to the 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 subcircuit is the largest one (in terms of the 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 labeled 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. Fig. 10(a) shows an example of a cut, enclosed in the blue dashed line. Each node in the graph is labeled with its significance. The cut significance is defined as the sum of all its output nodes significance: in the example, $\sigma_{C_1} = \sigma_{n_4} + \sigma_{n_2} = 8 + 4 = 12$.

Fig. 10(b) 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 node subsets that can be considered as the best candidates for approximation. The red highlighted path corresponds to cut C_1 in Fig. 10(a): indeed, 1-branch is taken for nodes n_2 and n_4 only. The algorithm identifies the largest cut whose significance does not overcome the predefined error threshold T.

Since the worst case complexity of the exploration is exponential in the number of nodes, three criteria are employed to bind 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 a possible solution: if a branch represents a nonclosed 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.* [45], 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 toward better candidates for ALS.

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:

1) logic rewriting by the Boolean optimization;
2) logic rewriting by the Boolean matrix factorization (BMF);
3) logic rewriting by the binary decision diagrams (BDDs);
4) 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 shown in Fig. 11. Depending on the error metric, the QoR circuit can compute the arithmetic difference or the 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

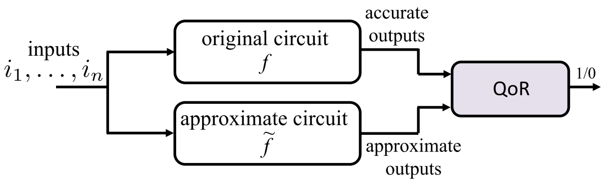


Fig. 11. *In SALSA, a QoR circuit is constructed to compare the outputs of exact and approximate circuits [57]. Observability don't cares of the approximate circuit are used to minimize the approximate circuit logic.*

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

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.

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 1. 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] proposed 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 rewritten 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 a different impact on the error and circuit complexity, as measured by the design area. Since a circuit can have millions of internal expressions, each with a number of possible ways to rewrite 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 the 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) underweight constraints (i.e., maximum error).

B. Logic Rewriting Using Boolean Matrix Factorization

BLASYS introduces a new formal method in ALS [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 an 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, BMF is used, where an input matrix M of dimensions $N \times p$ is factored into two matrices: an $N \times d$ matrix B and a $d \times p$ matrix C, where $1 \leq d < p$ such that $|M - BC|_2$ is minimized, that is, $M \approx BC$ [32]. In 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 the following equation[2]:

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

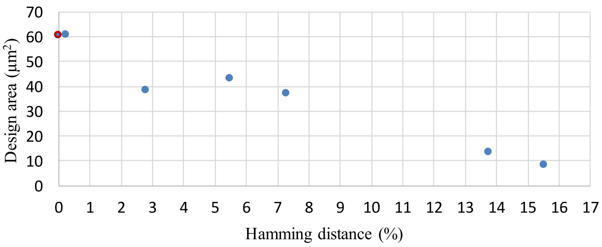


Fig. 12. *Tradeoff between the design area and QoR as measured by HD for x2 circuit. The original design point is in red and the approximate designs in blue. Approximate designs are obtained by adjusting the factorization degree in BLASYS from 6 down to 1.*

Then, for the case of $d = 2$, M can be factored as given in the following equation:

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

leading to errors in the approximation, which is equal to 4 as measured by the 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 typically leads to a synthesized circuit with less design area and power consumption. Fig. 12 shows an example of the tradeoff between QoR and design area where BLASYS is used to synthesize approximate versions for circuit x2 from the LGSynth91 benchmark set [63]. x2 has ten primary inputs and seven primary outputs; thus, one can vary the factorization degree d from $d = 6$ down to $d = 1$. By changing d, BLASYS enables a graceful tradeoff between QoR and design area. For example, for x2, we can reduce the 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

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

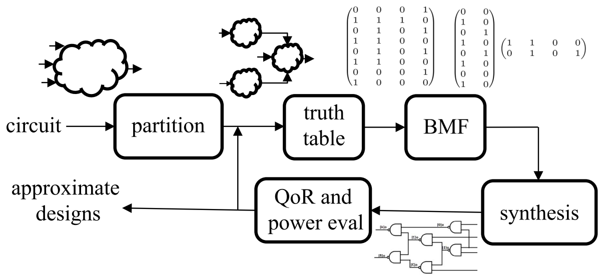


Fig. 13. ***BLASYS flow [30]. An input circuit is first partitioned into subcircuits with a reasonable number of inputs for each subcircuit. The truth table of every subcircuit is then evaluated followed by BMF. The results from BMF are then synthesized to create an approximate subcircuit. The QoR and 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 tradeoffs.***

of subcircuits, as shown in Fig. 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.

C. Logic Rewriting Using Binary Decision Diagrams

Reduced-order BDDs 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, that is, 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. Fig. 14(a) shows 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, that is, 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, that is, cofactors. For example, the BDD in Fig. 14(b) is obtained from the BDD in Fig. 14(a) 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, that is, the child with a fewer number of assignments that lead to the terminal 1, is chosen. For example, the approximate BDD in Fig. 14(c) is obtained from the BDD in Fig. 14(a) 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 original circuit [13]. To identify this function, a new BDD, g, is constructed such that it enumerates every possible function whose output differs in at most e bits compared with the original circuit. That is, g represents every potential circuit approximation with at most e output flips. For example, consider a single-output circuit $f(i_1, i_2)$ with two primary inputs (i_1, i_2); then, it has four output possibilities for each one of the possible four input combinations $\{i_1 i_2 = 00, i_1 i_2 = 01, i_1 i_2 = 10, i_1 i_2 = 11\}$. A new BDD $g(b_1, b_2, b_3, b_4, i_1, i_2)$ is constructed with four additional variables $\{b_1, \ldots, b_4\}$ that are created to indicate which input combinations will have an output bit flip, that is, if $b_j = 1$ for $j \in \{1, \ldots, 4\}$ then $g(b_1, b_2, b_3, b_4, i_1, i_2) \neq f(i_1, i_2)$; otherwise, $g(b_1, b_2, b_3, b_4, i_1, i_2) = f(i_1, i_2)$. In the BDD of g, a partial path that assigns values for b_1, b_2, b_3, b_4 will lead to 0 if the output function of the approximate circuit has more than e bit flips; otherwise, it will lead to a subgraph BDD, whose variables are the two original circuit inputs. 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 with the original BDD, we will find that no more than e paths lead to different outcomes.

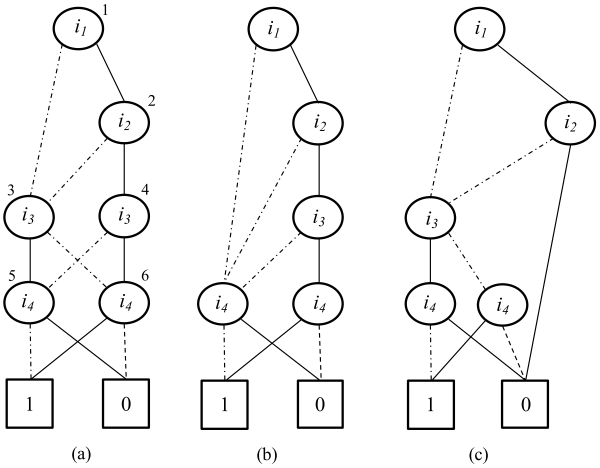
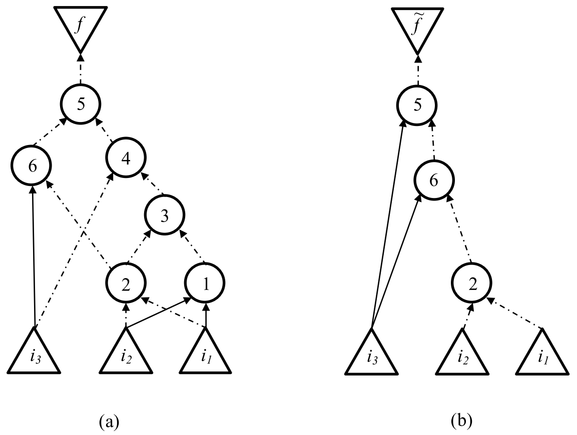


Fig. 14. ***Examples of approximate Boolean rewriting using BDDs. (a) Original BDD. (b) and (c) Approximate BDD.***



(a) (b)

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. (a) Original AIG. (b) Approximate AIG.***

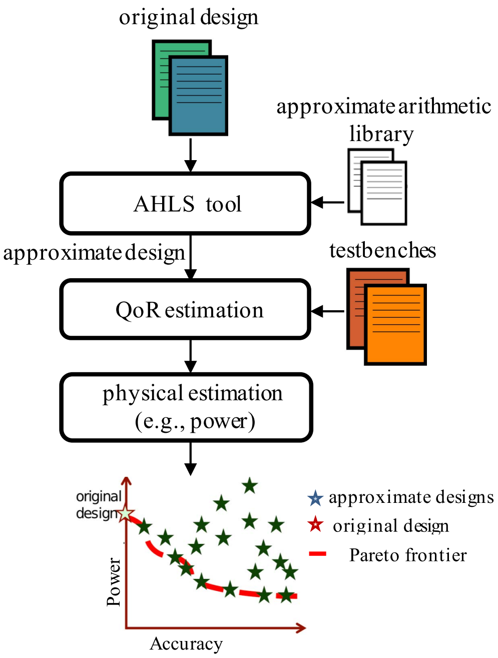


Fig. 16. ***General flow of AHLS methods.***

D. Logic Rewriting Using AND-Inverter Graphs

An AND-Inverter Graph (AIG) is a graphical representation for circuits where nodes correspond to two-input AND gates and edges can be either inverted or not inverted [33]. For example, Fig. 15 shows the circuits represented in AIGs where dashed edges represent inverted signals and i_1, i_2, and i_3 represent the primary input signals and f represents the primary output signal. AIGs provide a scalable graphical representation for circuit synthesis; however, unlike BDDs, they are not canonical. Using AIGs as the Boolean representation, Chandrasekharan *et al.* [5] proposed an algorithm for approximate AIG rewriting 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 Fig. 15(a), 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 Fig. 15(a), 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, that is, 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 Fig. 15(a), then the approximate AIG in Fig. 15(b) is obtained. A SAT solver is then employed to compare the original AIG and the approximate AIGs to check whether the error constraint is violated, hence guaranteeing the error bound.

V. APPROXIMATE HIGH-LEVEL SYNTHESIS

All the aforementioned ALS techniques operate on either the gate-based netlist or on a Boolean representation of the circuit. In contrast, the goal of 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. Similar 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 an outcome, as shown in Fig. 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, that is, 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 tradeoff between QoR and 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 shown in Fig. 17. A number of transformation operators are then

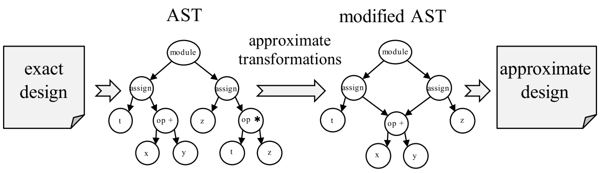


Fig. 17. ***ABACUS approximation flow. An input design is parsed and captured in an AST form. The AST is then manipulated to generate approximate ASTs that are written back to Verilog and synthesized into approximate circuits.***

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 the 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.
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 of saving one multiplier.
5) *Loop transformations:* In ABACUS, all Verilog loops are unrolled, and the statements in the unrolled loop are then approximated using the aforementioned transformations.

An attractive feature of ABACUS is that it applies the transformation operators in a random fashion and then selects and retains the approximate designs that have the optimal tradeoff 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], and [68]) propose a more detailed analysis of 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 tradeoff for addition. To choose the best approximate adder, different approximate adders are substituted in the circuit, 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 the power-QoR tradeoff, then the approximate design is accepted. To determine the best precision, Constantinides *et al.* [6] formulated 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 toward approximation. It is generalized by Li *et al.* [25], which also considers 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 *et al.* [24] proposed 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 the 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 is chosen to minimize the latency or energy subject to QoR constraints.

Roldao-Lopes *et al.* [41] also proposed 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 tradeoffs between resources and accuracy.

VI. COMPARATIVE EVALUATION OF ALS METHODS

As described in Sections III and IV, ALS methods proposed in the literature have 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 ALS (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 1 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. The 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, performs 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 Figs. 18 and 19, we plot the relative area and mean absolute error (%) of the 8- and 16-bit multipliers in comparison to the accurate multipliers. We also plot the Pareto frontier for the designs with an optimal tradeoff between the design area and MAE. The plot leads to the following insights.

1) 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).
2) For the 16-bit multiplier, BLASYS shows better results as it is able to dominate the Pareto frontier. These better results show that BLASYS has better scalability since it is able to partition the circuit for larger designs and then rewrite the logic for each partition. GLP is dominated except for small values of MAE when it tends to produce reasonable results.
3) BLASYS (and EvoApprox for 8-bit multiplier) dominates 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 a 45% area reduction compared with an 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 labor-intensive and more flexible.

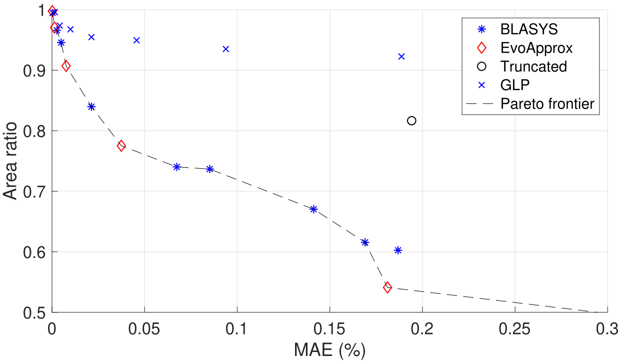


Fig. 18. *Approximate design variants of an 8-bit unsigned multiplier. Approximate designs were created with BLASYS, EvoApprox, GLP, and manually by truncating the least significant bits.*

A second comparison is provided in Fig. 20; here, the focus was set on containing the worst case error using ALS methods. We considered two structural transformation techniques—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

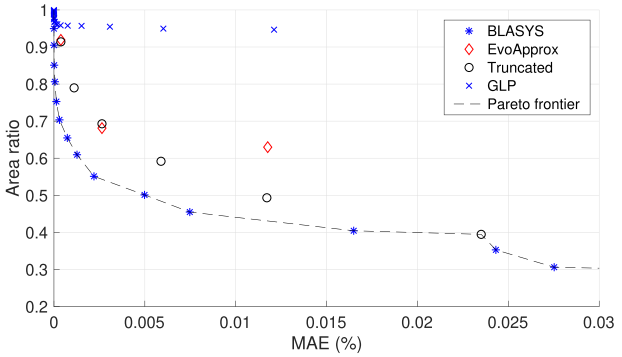


Fig. 19. *Approximate design variants of a 16-bit unsigned multiplier. Approximate designs were created with BLASYS, EvoApprox, GLP, and manually by truncating the least significant bits.*

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

Table 1 Characteristics of 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

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. Fig. 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 eight-point FFT benchmark written in Verilog RTL. We contrast ABACUS [38], which is an example of approximate HLS tools, against BLASYS [19] that relies on approximate logic rewriting.[5] For BLASYS, the circuit is first compiled to Boolean representation using Yosys. Fig. 21 shows the error in mse (%) versus area ratio compared with 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 a smoother tradeoff between accuracy and area as it incrementally approximates each partition in the design. However, this fine granularity comes at an expense:

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

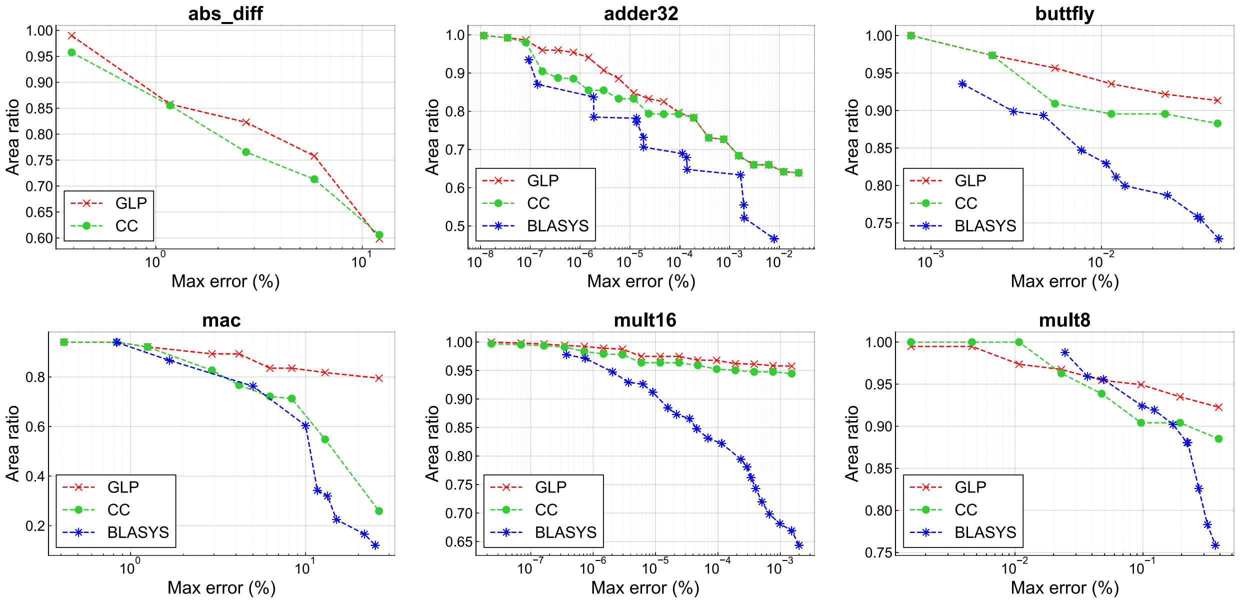


Fig. 20. ***Comparison of area reduction in approximate circuits for GLP [45], CC [44], and BLASYS [19]. The maximum absolute error (max error) is expressed as a percentage of the maximum circuit output.***

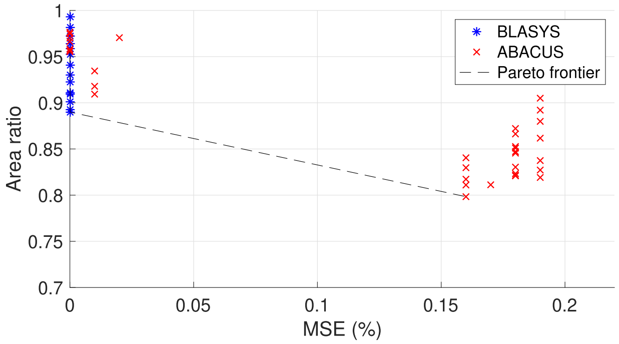


Fig. 21. ***Approximate design variants of an eight-point FFT circuit created using ABACUS and BLASYS.***

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 offer much better scalability in comparison with logic rewriting methods.

We summarize our insights from all experiments.

1) Structural transformation ALS methods require a synthesized circuit netlist as input before any AT is 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, for example, in gate-elimination-only methodologies. On the other hand, logic rewriting approaches do not assume a synthesized circuit, and the approximation is either conducted before or in conjunction with the synthesis process. Thus, logic rewriting can potentially lead to a larger design space exploration since the synthesis is part of the process. However, due to the heuristic nature of the synthesis process, logic rewriting 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 rely on underlying scalable circuit data structures, such as AIGs, or use partitioning techniques to break down the circuit into manageable subcircuits where logic rewriting 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, e.g., BLASYS), and others are instead built specifically for one given metric (such as, e.g., CC that is guaranteed to never exceed a maximum error but cannot be easily adapted to average error).
5) Compared with 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 circuit variants that have incremental nature with fine grain tradeoff 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 tradeoff. 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 rewriting ALS methods. An additional advantage for HLS methods over other methods is that their approximate designs are easier to interpret by human designers.

VII. CONCLUSION AND OPEN CHALLENGES IN ALS

ALS 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 the applications are data-rich with a high degree of error tolerance. Table 2 provides a taxonomy summary of the majority of articles 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 the low-level design of approximate components and methodologies for AHLS. We discuss the main challenges for ALS in the remainder of this section.

A. Synthesis of Approximate Circuits

1) 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 a 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.

2) Runtime Accuracy-Configurable Hardware: An open research question regards the generation of designs having an approximate degree, 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 a 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]

Table 2 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

have been proposed, their design is still manually done. Automated approaches that are able to explore the benefits of accuracy configuration, in light of the overhead (energywise and areawise) implied by the logic governing the transitions between different approximate settings are, instead, currently lacking.

B. Approximate High-Level Synthesis

1) 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 the existing studies on approximate arithmetic have been focused on integer arithmetic disregarding floating-point operators. Building robust AHLS tools requires extensive libraries of approximate floating- and 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 realize 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. Therefore, solutions are restricted only to integrating the available library elements. More flexible approaches have indeed been proposed, but they only focus on signal width optimizations [7], [8], without considering the opportunities for logic simplification offered by ALS. These are, instead, leveraged by the strategies in [2] and [20], which employs 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 (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 and SSIM).

2) 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* fashion, 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 toward the maturity of AHLS will be the development of strategies to quickly link error and QoR metrics across abstraction levels. On the one hand, 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 as approximate neural networks [55]. On the other hand, methodologies should be envisioned that estimate the QoR impact of a simplifying transformation, 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 the classification accuracy or SNR, while, at the operator levels, ERs 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 Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 4, pp. 1352–1361, Apr. 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 Architectures*. New York, NY, USA: Springer, Mar. 2020, pp. 14–29.

[3] B. Barrois, O. Sentieys, and D. Menard, "The hidden cost of functional approximation against careful data sizing—A case study," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 181–186.

[4] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[5] A. Chandrasekharan, M. Soeken, D. Große, and R. Drechsler, "Approximation-aware rewriting of AIGs for error tolerant applications," in *Proc. 35th Int. Conf. Comput.-Aided Des.*, Nov. 2016, p. 83.

[6] G. A. Constantinides, P. Y. K. Cheung, and W. Luk, "Optimum wordlength allocation," in *Proc. 10th Annu. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2002, pp. 219–228.

[7] G. Constantinides, A. Kinsman, and N. Nicolici, "Numerical data representations for FPGA-based scientific computing," *IEEE Des. Test. Comput.*, vol. 28, no. 4, pp. 8–17, Jul. 2011.

[8] G. A. Constantinides, P. Cheung, and W. Luk, "The multiple wordlength paradigm," in *Proc. IEEE Symp. FPGAs for Custom Comput. Mach.*, pp. 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 *Proc. Int. Conf. Parallel Problem Solving Nature*, 2000, pp. 849–858.

[10] T. A. Drane, T. M. Rose, and G. A. Constantinides, "On the systematic creation of faithfully rounded truncated multipliers and arrays," *IEEE Trans. Comput.*, vol. 63, no. 10, pp. 2513–2525, Oct. 2014.

[11] L. Ferretti *et al.*, "Tailoring SVM inference for resource-efficient ECG-based epilepsy monitors," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 1–4.

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

[13] S. Froehlich, D. Grosse, and R. Drechsler, "Error bounded exact BDD minimization in approximate computing," in *Proc. Int. Symp. Multi-Level Logic*, May 2017, pp. 254–259.

[14] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *Proc. IEEE Eur. TEST Symp. (ETS)*, May 2013, pp. 1–6.

[15] S. Hashemi, R. I. Bahar, and S. Reda, "DRUM: A dynamic range unbiased multiplier for approximate applications," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, Nov. 2015, pp. 418–425.

[16] S. Hashemi, R. I. Bahar, and S. Reda, "A low-power dynamic divider for approximate applications," in *Proc. IEEE/ACM Design Autom. Conf.*, vol. 105, Jun. 2016, pp. 1–6.

[17] S. Hashemi and S. Reda, "Generalized matrix factorization techniques for approximate logic synthesis," in *Proc. ACM/IEEE Design Automat. Test Eur.*, Mar. 2019, pp. 1289–1292.

[18] S. Hashemi, H. Tann, F. Buttafuoco, and S. Reda, "Approximate computing for biometric security systems: A case study on iris scanning," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 319–324.

[19] S. Hashemi, H. Tann, and S. Reda, "BLASYS: Approximate logic synthesis using Boolean matrix factorization," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, Jun. 2018, pp. 55:1–55:6.

[20] J. Huang, J. Lach, and G. Robins, "A methodology for energy-quality tradeoff using imprecise hardware," in *Proc. 49th Annu. Design Autom. Conf. (DAC)*, Jun. 2012, pp. 504–509.

[21] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *Proc. DAC Design Autom. Conf.*, 2012, pp. 820–825.

[22] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *Proc. 24th Int. Conf. VLSI Design*, Jan. 2011, pp. 346–351.

[23] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. ACM Int. Symp. Code Gener. Optim.*, 2004, pp. 75–84.

[24] S. Lee, L. K. John, and A. Gerstlauer, "High-level synthesis of approximate hardware under joint precision and voltage scaling," in *Proc. IEEE/ACM Design Automat. Conf.*, Mar. 2017, pp. 187–192.

[25] C. Li, W. Luo, S. S. Sapatnekar, and J. Hu, "Joint precision optimization and high level synthesis for approximate computing," in *Proc. IEEE/ACM Design Autom. Conf.*, vol. 104, Jun. 2015, pp. 1–6.

[26] A. Lingamneni, C. Enz, K. Palem, and C. Piguet, "Synthesizing parsimonious inexact circuits through probabilistic design techniques," *ACM Trans. Embedded Comput. Syst.*, vol. 12, no. 2s, pp. 1–26, May 2013.

[27] G. Liu and Z. Zhang, "Statistically certified approximate logic synthesis," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2017, pp. 344–351.

[28] W. Liu, L. Chen, C. Wang, M. O'Neill, and F. Lombardi, "Design and analysis of inexact floating-point adders," *IEEE Trans. Comput.*, vol. 65, no. 1, pp. 308–314, Jan. 2016.

[29] W. Liu, F. Lombardi, and M. Shulte, "A retrospective and prospective view of approximate computing [Point of view]," *Proc. IEEE*, vol. 108, no. 3, pp. 394–399, Mar. 2020.

[30] J. Ma, S. Hashemi, and S. Reda, "Approximate logic synthesis using BLASYS," in *Proc. Workshop Open-Source EDA Technol.*, no. 5, 2019, pp. 1–3, Art. no. 5.

[31] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *Proc. Int. Conf. Comput. Aided Design*, Nov. 2013, pp. 779–786.

[32] P. Miettinen and J. Vreeken, "Model order selection for Boolean matrix factorization," in *Proc. 17th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2011, pp. 51–59.

[33] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *Proc. IEEE/ACM Design Autom. Conf.*, Jul. 2006, pp. 532–535.

[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 *Proc. IEEE/ACM Design Autom. Conf.*, vol. 123, Jun. 2019, pp. 1–6.

[35] V. Mrazek, Z. Vasicek, and L. Sekanina, "EvoApproxLib: Extended library of approximate arithmetic circuits," in *Proc. Workshop Open-Source EDA Technol.*, no. 10, 2019, pp. 1–4.

[36] K. E. Murray, A. Suardi, V. Betz, and G. Constantinides, "Calculated risks: Quantifying timing error probability with extended static timing analysis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 4, pp. 719–732, Apr. 2019.

[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. Emerg. Topics Comput.*, vol. 7, no. 1, pp. 18–30, Jan. 2019.

[38] K. Nepal, Y. Li, R. I. Bahar, and S. Reda, "ABACUS: A technique for automated behavioral synthesis of approximate computing circuits," in *Proc. IEEE/ACM Design, Autom. Test Eur.*, Mar. 2014, pp. 1–6.

[39] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan, "ASLAN: Synthesis of approximate sequential circuits," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, Mar. 2014, pp. 1–6.

[40] S. Rehman, W. El-Harouni, M. Shafique, A. Kumar, and J. Henkel, "Architectural-space exploration of approximate multipliers," in *Proc. Int. Conf. Comput. Aided Design*, Nov. 2016, pp. 1–8.

[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 *Proc. 17th IEEE Symp. Field-Program. Custom Comput. Mach.*, Apr. 2009, pp. 209–216.

[42] H. Saadat, H. Javaid, and S. Parameswaran, "Approximate integer and floating-point dividers with near-zero error bias," in *Proc. 56th Annu. Design Autom. Conf.*, Jun. 2019, pp. 161:1–161:6.

[43] I. Scarabottolo, G. Ansaloni, G. Constantinides, and L. Pozzi, "Partition and propagate: An error derivation algorithm for the design of approximate circuits," in *Proc. 56th Design Autom. Conf.*, Jun. 2019, pp. 1–6.

[44] I. Scarabottolo, G. Ansaloni, and L. Pozzi, "Circuit carving: A methodology for the design of approximate hardware," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, Mar. 2018, pp. 545–550.

[45] J. Schlachter, V. Camus, K. V. Palem, and C. Enz, "Design and applications of approximate circuits by gate-level pruning," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 5, pp. 1694–1702, May 2017.

[46] E. Sentovich and K. Singh, "A system for sequential circuit synthesis," EECS, UCB, Berkeley, CA, USA, Tech. Rep. UCB/ERL M92/41, 1992.

[47] M. Shafique, W. Ahmad, R. Hafiz, and J. Henkel, "A low latency generic accuracy configurable adder," in *Proc. 52nd Design Autom. Conf.*, Jul. 2015, pp. 1–6.

[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 *Proc. Design Autom. Conf.*, 2014, pp. 1–6.

[49] D. Shin and S. K. Gupta, "A new circuit simplification method for error tolerant applications," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, Mar. 2011, pp. 1–6.

[50] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proc. SIGSOFT Symp. Eur. Conf. Found. Softw. Eng.*, Sep. 2011, pp. 124–134.

[51] M. Soeken, D. Große, A. Chandrasekharan, and R. Drechsler, "BDD minimization for approximate computing," in *Proc. 21st Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2016, pp. 474–479.

[52] J. E. Stine *et al.*, "FreePDK: An open-source variation-aware design kit," in *Proc. IEEE Int. Conf. Microelectron. Syst. Edu. (MSE)*, Jun. 2007, pp. 173–174.

[53] S. Su, Y. Wu, and W. Qian, "Efficient batch statistical error estimation for iterative multi-level approximate logic synthesis," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, Jun. 2018, pp. 54:1–54:6.

[54] Z. Vasicek and L. Sekanina, "Evolutionary approach to approximate digital circuits design," *IEEE Trans. Evol. Comput.*, vol. 19, no. 3, pp. 432–444, Jun. 2015.

[55] F. Vaverka, V. Mrazek, Z. Vasicek, and L. Sekanina, "TFApprox: Towards a fast emulation of DNN approximate hardware accelerators on GPU," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, Mar. 2020, pp. 1–6.

[56] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, Mar. 2013, pp. 1367–1372.

[57] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "SALSA: Systematic logic synthesis of approximate circuits," in *Proc. 49th Design Autom. Conf.*, Jun. 2012, pp. 796–801.

[58] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan, "MACACO: Modeling and analysis of circuits for approximate computing," in *Proc. Int. Conf. Comput. Aided Design*, Nov. 2011, pp. 667–673.

[59] C. Wolf. *Yosys Open SYnthesis Suite*. Accessed: 2016. [Online]. Available: http://www.clifford.at/yosys/documentation.html

[60] Y. Wu and W. Qian, "An efficient method for multi-level approximate logic synthesis under error rate constraint," in *Proc. 53rd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2016, pp. 1–6.

[61] Q. Xu, T. Mytkowicz, and N. Kim, "Approximate computing: A survey," *IEEE Des. Test*, vol. 33, no. 1, pp. 8–22, Jan. 2016.

[62] C. Yang, M. Ciesielski, and V. Singhal, "BDS: A BDD-based logic optimization system," in *Proc. ACM/IEEE Design Autom. Conf.*, Jul. 2000, pp. 866–876.

[63] S. Yang, "Logic synthesis and optimization benchmarks user guide," Microelectron. Center North Carolina, Research Triangle, NC, USA, 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 *Proc. IEEE Int. Conf. Comput. Design (ICCD)*, Nov. 2017, pp. 517–524.

[65] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu, "On reconfiguration-oriented approximate adder design and its application," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2013, pp. 48–54.

[66] P. Yin, C. Wang, W. Liu, and F. Lombardi, "Design and performance evaluation of approximate floating-point multipliers," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, Jul. 2016, pp. 296–301.

[67] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: From error visibility to structural similarity," *IEEE Trans. Image Process.*, vol. 13, no. 4, pp. 600–612, Apr. 2004.

[68] G. Zervakis, S. Xydis, D. Soudris, and K. Pekmestzi, "Multi-level approximate accelerator synthesis under voltage island constraints," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 66, no. 4, pp. 607–611, Apr. 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 *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–4.

[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 Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 18, no. 8, pp. 1225–1229, Aug. 2010.

ABOUT THE AUTHORS

Ilaria Scarabottolo (Member, IEEE) received the B.Sc. degree in mathematical engineering and the M.Sc. degree in computer engineering from the Politecnico di Milano, Milan, Italy, in 2013 and 2016, respectively, and the M.Sc. degree from École Centrale Paris, Châtenay-Malabry, France, in 2014. She is working toward the Ph.D. degree at the Faculty of Informatics, Università della Svizzera Italiana (USI), Lugano, Switzerland.

Her research interests include approximate logic synthesis for error-tolerant applications, embedded systems, and low-power hardware design.

Ms. Scarabottolo was awarded the Swiss National Foundation Grant DOC-Mobility in 2019, because of which she spent six months as a Visiting Student at Imperial College London.

Giovanni Ansaloni (Member, IEEE) received the M.Sc. degree in electronic engineering from the University of Ferrara, Ferrara, Italy, in 2003, the M.A.S. degree from the Advanced Learning and Research Institute (ALaRI), Lugano, Switzerland, in 2005, and the Ph.D. degree from the Università della Svizzera Italiana (USI Lugano), Lugano, in 2011.

From 2011 to 2015, he was a Researcher with the École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland. He is currently a Postdoctoral Researcher with the Faculty of Informatics, USI Lugano. His research efforts cover different areas related to hardware/software codesign, including high-level synthesis, domain-specific architectures, and low-power signal processing.

George A. Constantinides (Senior Member, IEEE) received the Ph.D. degree from Imperial College London, London, U.K., in 2001.

Since 2002, he has been with the Faculty with Imperial College London, where he is currently a Professor of digital computation and the Head of the Circuits and Systems Research Group. He has published over 200 research articles in peer-refereed journals and international conferences.

Prof. Constantinides is a Fellow of the British Computer Society. He also serves on several program committees. He was the General Chair of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays in 2015.

Laura Pozzi (Member, IEEE) received the Ph.D. degree in computer engineering from the Politecnico di Milano, Milan, Italy, in 2000.

She was a Postdoctoral Researcher with the École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, a Research Engineer with STMicroelectronics, San Diego, CA, USA, and an Industrial Visitor with the University of California at Berkeley, Berkeley, CA, USA. She is currently a Professor with the Faculty of Informatics, Università della Svizzera Italiana, Lugano, Switzerland. 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 of Integrated Circuits and Systems and *IEEE Design & Test of Computers*. She has been on the technical program committee of several international conferences in the areas of compilers and architectures for embedded systems.

Sherief Reda (Senior Member, IEEE) received the Ph.D. degree in computer science and engineering from the University of California at San Diego, San Diego, CA, USA, in 2006.

He is currently a Full Professor with the School of Engineering, Brown University, Providence, RI, USA. He has over 120 publications in peer-reviewed conferences and journals with several of them receiving best paper nominations and awards. His research interests include energy-efficient computing, thermal-power sensing and management, low-power design techniques, and design automation.

Dr. Reda serves as an Associate Editor for the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.