A Formal Framework for Maximum Error Estimation in Approximate Logic Synthesis

Ilaria Scarabottolo, Giovanni Ansaloni, George A. Constantinides and Laura Pozzi

***Abstract*—Approximate Logic Synthesis techniques have become popular in error-resilient systems, where accuracy requirements can be traded for improved energy efficiency. Many of these techniques operate on a circuit by substituting or removing some of its portions under a predefined error constraint; however, research on systematic methods to determine the error induced by such transformations is still at an early stage. We propose herein a generic framework for modeling maximum error in a circuit, called Partition and Propagate, which is a fundamental preliminary step for ALS. This framework is based on circuit partitioning and error propagation among the sub-circuits. We provide a sound, complete formal description of such framework, and we illustrate how two state-of-the-art algorithms can be subsumed by it. Moreover, we propose a novel gate-level error-modeling algorithm which is able to identify the whole range of possible errors induced by a given approximate transformation. We compare the three strategies and illustrate the efficiency of the new error-propagation methodology, which is able to identify accurate error bounds and, hence, guide ALS techniques to more valuable solutions.**

***Index Terms*—approximate computing, logic synthesis, efficient architecture, hardware design, error modeling.**

I. Introduction

Energy efficiency is nowadays regarded as a most crucial concern in a large spectrum of applications, ranging from portable devices to cloud computing and data warehouses. Approximate Computing leverages the inherent error resilience of many real-world scenarios to improve scalability and reduce energy consumption at the expense of a slight reduction in output accuracy [1].

In particular, approximate circuits provide a solution for improving hardware performance (such as area, latency and power consumption) by relaxing the requirement for fully accurate computation. The process of designing approximate circuits starting from their high-level description is called Approximate Logic Synthesis (ALS), and has recently attracted many research efforts.

A large family of ALS methods operate on a gate-level netlist representing a circuit and identify portions of the circuit that can be neglected [2], [3], or substituted [4], without impacting too strongly on the final result quality. The efficiency of these techniques relies deeply on the availability of an accurate circuit error model. In other words, it is of crucial importance to possess an accurate estimate of the effect that a given approximate transformation of the original circuit will have.

Although other works [5], [6], [7] have provided an initial approach to error estimation, the literature lacks a sound formalisation of the problem, along with generic, automatic and efficient algorithms to solve it.

This work aims at addressing this gap by providing a sound description of the error modeling framework. In particular, it focuses on providing *bounds on the maximum error* entailed by a given approximate transformation.

The principal contributions of the paper are:

- a formal definition of error model of a circuit described at gate-level;
- the formal description of Partition and Propagate, an efficient framework to derive such error model and, in particular, to identify bounds on the errors induced by approximate transformations, through circuit partition;
- the presentation of a novel algorithm fitting the above-mentioned framework, as well as the illustration of two state-of-the-art algorithms that follow it.

The next section introduces the state of the art in ALS, further motivates the necessity of a formal framework description and outlines the weaknesses of currently available approaches. Section III provides an overview of the proposed framework, which is then formally defined in Section IV and V. Section VI illustrates how two previously-published approaches are subsumed by our framework. Section VII instead describes a new algorithm fitting the framework, proposed in this work. Finally, Section VIII illustrates the effectiveness of the presented strategy in guiding ALS methods through experimental evaluation.

II. Motivation

Approximate Computing (AC) as a design paradigm has been attracting a large amount of research effort. While approximation can be exploited at various levels of the hardware/software stack [1], [8], circuit-level AC methodologies are most related to our contribution. In particular, we focus on Approximate Logic Synthesis, which consists of manipulating the Boolean function implemented by a circuit to obtain an inexact counterpart, as opposed to Voltage Overscaling, where the voltage supply of an architecture is altered, injecting timing errors [9], [10].

Some notable efforts in inexact circuit research focus on manually designing specific arithmetic units, such as adders [11] or multipliers [12], [13], while others adopt a more generic approach, enabling the simplification of any combinatorial circuit [2], [3], [4], [7], [14], [15], [16]. A wide variety

M. Shell was with the Department of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, 30332 USA e-mail: (see http://www.michaelshell.org/contact.html).

J. Doe and J. Doe are with Anonymous University.

Manuscript received April 19, 2005; revised August 26, 2015.

of techniques belong to this category, including methods exploiting BDDs [17], [18] or and-inverter graphs [19], as well as combining together approximate units to realise more complex systems [20], [21], [22].

A large portion of these ALS methods is based on the derivation of inexact circuits by the elimination (or substitution) of some components from the original ones [2], [3], [4], [23], [24], [25]. Some of these techniques initially blindly apply such transformations and then evaluate the effect that they have on the output. Other methods instead, in order to choose which components are the best candidates for elimination, undergo the preliminary step of assigning each component a value. This value, or *weight*, represents an estimate of the error that its removal can induce on the final output.

Figure 1 illustrates this process: the original circuit, expressed as a gate-level netlist, undergoes an error modeling phase, the output of which is the circuit labelled with weights. With this additional information, the ALS method itself is applied, leading to the synthesis of the desired approximate circuit, which has to respect some pre-defined error constraints. Finally, an error validation phase aims at verifying that the given error threshold has not been violated.

The strategies adopted so far to obtain such weights are of three different types: Monte Carlo sampling of the circuit inputs and simulation over the resulting input subset is a first possibility [4], [7], [15]. However, the accuracy of the weights obtained through this strategy relies entirely on the size of the sample and, most importantly, the obtained results do not guarantee any bound on the induced error.

Other works exhaustively evaluate such weights, either explicitly by fully simulating the circuit [3], or implicitly by employing SAT-solvers [6]. While these strategies derive exact weights, they clearly present scalability problems when applied to large circuits, since their complexity is exponential in the number of the circuit inputs. Indeed, in [3], full circuit simulation is employed for small benchmarks, while gate-level errors for large benchmarks have been derived inductively on identical circuit blocks.

Finally, conservative bounds can be adopted to estimate such weights, as in [2], where node weights are assigned to the sum of the significance of all their reachable primary outputs. However, such overly conservative weights have proven to provide poor guidance to the ALS method applied subsequently [5]. The strengths and weaknesses of the methods described above are summarised in Figure 2.

In this work we provide a complete framework for circuit maximum error modeling, Partition and Propagate, which allows for a controlled trade-off between QoR (*i.e.*, the accuracy of the obtained weights) and execution time. Actually, conservative bounds estimation and full simulation are particular instances of algorithms subsumed in such framework, representing the two extreme points. We will also present an algorithm lying between these extremes, which overcomes the limitations of the previously described approaches by allowing for a parameterizable trade-off between execution time and accuracy, while maintaining bound guarantees.

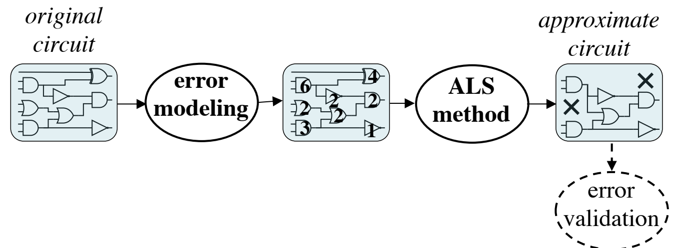

Fig. 1: The original circuit is labelled with weights through an error modeling phase, then the ALS method is applied to synthesize an approximate circuit. Finally, the error constraint is checked through error validation.

	accuracy	***scalability***	***bounds guarantee***
Exact evaluation	✓	✗	✓
Conservative bounds	✗	✓	✓
Monte Carlo sampling	?	✓	✗

Fig. 2: Summary of the main strengths and weaknesses of the state of the art techniques for error modeling.

III. Partition and Propagate overview

In Partition and Propagate (P&P), the circuit is represented as a Directed Acyclic Graph (DAG), such as the one in Figure 3. The purpose of this work is to derive an error-model, where we aim at calculating bounds on the maximum error induced on the circuit output if a gate is removed from the circuit. A simple example of such model is provided in Figure 3: each node is labelled with an integer, called its *weight*, which represents a bound for the corresponding gate. In this example, the output represents a 4-bit binary number, so the output weights are set to their bit-significance (consecutive powers of two), while the weights of all other nodes are derived through the approach presented in this paper.

The crucial step we introduce to control the execution time of error-modeling is to partition the graph into subgraphs: the reduced size of these subgraphs allows their exhaustive simulation, so that bounds on maximum error can be derived locally, by observing changes in each subgraph output. These bounds are then propagated among the different subgraphs, in a traverse from the primary outputs to the primary inputs, through a propagation procedure which we describe in depth in Section IV.

Figure 4 illustrates the P&P process: starting from the original circuit, where primary output nodes are labelled with their bit-significance, the graph is partitioned (step 1). In step 2, the partition-graph is traversed from the POs to the PIs, identifying propagation functions for each subgraph and transferring information from their outputs to their inputs. After this step, all bounds on subgraph output nodes are known. In step 3, bounds for internal nodes are derived through exhaustive simulation in each (small) subgraph.

The employment of full simulation on each subgraph separately guarantees scalability, while accuracy on estimated bounds is preserved thanks to the choice of a convenient

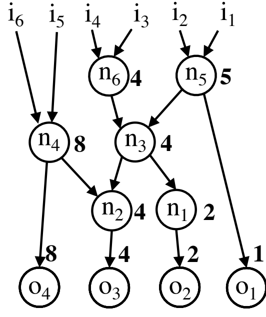

Fig. 3: A simple example of DAG labelled with weights. Primary output weights are assigned to the arithmetic bit significance.

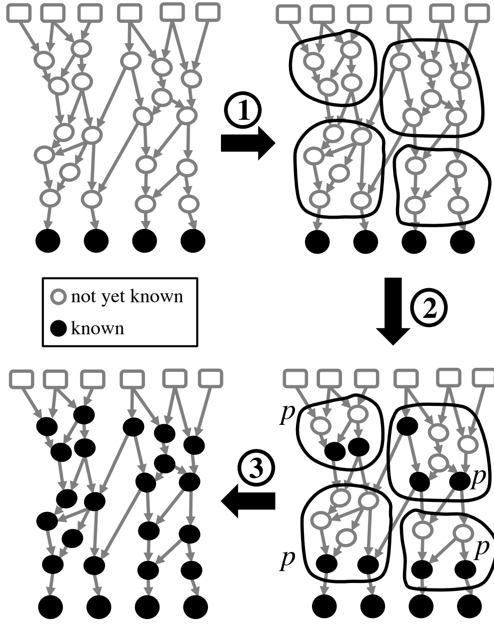

Fig. 4: Different phases of Partition and Propagate. Initially, only output bounds are assigned, then the graph is partitioned (1), the propagation function is applied for all subgraphs, so that subgraph output nodes are labelled (2) and, finally, full simulation is performed in every subgraph to label internal nodes (3).

propagation function. For graph partitioning, we employ the technique described in [5].

The next section will provide a formal definition of error model.

IV. Error model definition

As introduced in Section III, a combinatorial circuit can be represented as a Directed Acyclic Graph (N, E), where each node $n_i \in N$ represents a single-output Boolean gate and each edge $(n_i, n_j) \in E$ represents a connection between nodes such that the output of n_i is used by n_j. A graph (N, E) has a set of primary inputs I and a set of primary outputs O.

Each element of the input set I is an h-dimensional Boolean vector $\boldsymbol{x} \in \mathbb{B}^h$, and each element of the output set O is a k-dimensional Boolean vector $\boldsymbol{o} \in \mathbb{B}^k$, so that $|I| = 2^h$ and $|O| = 2^k$.

The result of a circuit computation is captured by function

$$f : \mathbb{B}^k \to \mathbb{D} \tag{1}$$

mapping the Boolean vector of the circuit primary outputs into any linearly ordered group, representing the value computed by the circuit. For example, if the output is an integer binary word, we may take $\mathbb{D} = \mathbb{Z}$, the integers, with f corresponding to the bit-significance weighting of the k-bit vector.

The purpose of error modeling is to obtain bounds on the influence of a node n_i on the circuit output, in terms of difference from the exact result that can be observed if n_i is removed from the circuit and its output is set to a constant value. For the rest of this section, we will assume that the node output is set to 0 without loss of generality, since the equivalent process can also be performed with the output set to 1.

Figure 5 represents three graphs which are identical, except for the value of n_i: the leftmost graph is the original one, where the value of node n_i is unknown. In the central graph, the node output is forced to 0, while in the rightmost graph it is forced to 1. All three graphs are fed with the same input $\boldsymbol{x} \in I$, and they generate three vectors $\boldsymbol{o}^{\boldsymbol{x}}$, $\boldsymbol{o}^{\boldsymbol{x}}_{i,0}$ and $\boldsymbol{o}^{\boldsymbol{x}}_{i,1}$ respectively. For each input $\boldsymbol{x}$, we are interested in error

$$e_{\boldsymbol{x}} = f(\boldsymbol{o}^{\boldsymbol{x}}) - f(\boldsymbol{o}^{\boldsymbol{x}}_{i,0}) \tag{2}$$

given by the difference between the exact output and the approximate one where the node in exam n_i has been forced to 0. Note that *we do not know* whether for that given input the node value was 0 or 1. However, we do know that the error is maximum if the node value is 1 in the exact circuit and, therefore, we bound $e_{\boldsymbol{x}}$ through

$$e_{01,\boldsymbol{x}} = f(\boldsymbol{o}^{\boldsymbol{x}}_{i,1}) - f(\boldsymbol{o}^{\boldsymbol{x}}_{i,0}) \tag{3}$$

since

$$|e_{\boldsymbol{x}}| \leq |e_{01,\boldsymbol{x}}| \tag{4}$$

We are interested in estimating the maximum possible discrepancy from the exact output, which is represented by the absolute value of error $e_{\boldsymbol{x}}$:

$$\max_{\boldsymbol{x}} |e_{\boldsymbol{x}}| \leq \max_{\boldsymbol{x}} |e_{01,\boldsymbol{x}}| = \max_{\boldsymbol{x}} |f(\boldsymbol{o}^{\boldsymbol{x}}_{i,1}) - f(\boldsymbol{o}^{\boldsymbol{x}}_{i,0})| \tag{5}$$

where the maximum is taken over all possible inputs $\boldsymbol{x} \in I$. Error modeling is, then, the process of estimating (5).

In the following sections, we will mainly refer to absolute error distance as the error metric for e in Eqs. (2)-(5). However, this approach can also handle other error metrics, such as, for instance, Hamming distance. In Eq. (1), let $\mathbb{D} = \mathbb{B}^k$, with f as the identity function; then define addition in the usual way as element-wise XOR for Equations (2) and (3) and, finally, define $|.|$ as the Hamming Weight, so that Eq. (4) is still valid. A straightforward change in Eq. (5) from $\max |e_{\boldsymbol{x}}|$ to $\mathbb{E}|e_{\boldsymbol{x}}|$

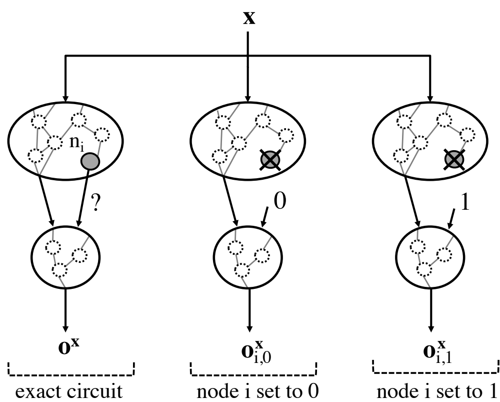

Fig. 5: Comparison between exact circuit and approximate versions with a gate set to 0 or 1.

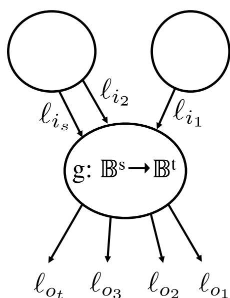

Fig. 6: Label propagation model for a generic subgraph with s inputs and t outputs, implementing function g.

for expected error, instead, would not be useful, since the bound of Eq. (4) would be too loose for a good estimate of the expected error. Indeed, the probabilities of nodes being zero or one should be taken into account, introducing significant modifications in the framework.

V. P&P Propagation technique

We define a *partition* function

$$P : (N, E) \mapsto S \tag{6}$$

where S is a set of subgraphs $(N_s, E_s),\ N_s \subseteq N,\ E_s \subseteq E$. Each node $n \in N$ is assigned to exactly one subgraph $(N_s, E_s) \in S$, and $\bigcup_s N_s = N$. Similarly, $E_s = \{(n_i, n_j) \in E \mid n_i \in N_s \wedge n_j \in N_s\}$.

We define *external edges* $\{(n_i, n_j) \mid n_i \in N_s \wedge n_j \in N_t \wedge s \neq t\}$ as those linking two different subgraphs. A subgraph (N_p, E_p) is *parent* of subgraph (N_q, E_q) if there exists at least one external edge (n_i, n_j) with $n_i \in N_p$ and $n_j \in N_q$.

Figure 6 illustrates a generic graph divided into three subgraphs, where the lower one has s inputs, t outputs and implements a generic function $g : \mathbb{B}^s \rightarrow \mathbb{B}^t$. Subgraph outputs are associated with a *label*, and a propagation function p is defined to derive labels of the s subgraph inputs, starting from the t subgraph outputs and function g:

$$p : L^t \times (\mathbb{B}^s \times \mathbb{B}^t) \longrightarrow L^s \tag{7}$$

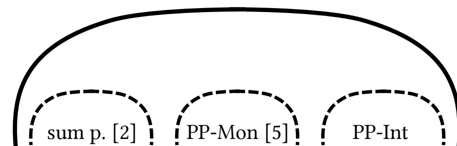

Fig. 7: Our formal framework subsumes different error-modeling approaches.

The purpose of P&P is to assign a label to each node $n \in N$, and these labels carry information on errors associated to approximate transformations. The label of an edge is equal to that of its destination node. The process of deriving labels through the propagation function is performed in a traverse of the graph from the POs to the PIs, starting from assigning the primary output labels (for example, to their bit significance, as in Figure 3). The definition of set L depends on the chosen error representation: for example, the integer weights introduced in Section III are particular instances of labels. We will see in Section VII that elements of L can also be intervals, providing an upper and lower bound to the maximum error entailed by a gate removal. The information contained in these labels is essential in boosting the efficiency of ALS methods, since knowledge on entailed error can guide the choice of applicable transformations.

Figure 7 illustrates that this framework captures a whole family of existing error-modelling approaches, each with different characteristics. In the following Section we show how two existing techniques, sum propagation [2] and P&P-Monotonicity (PP-Mon) [5], fit into the proposed framework. Section VII, instead, describes a novel algorithm called P&P-Intervals (PP-Int), which improves on the two state-of-the-art methods accuracy and simplicity.

VI. Existing Techniques

A. Sum propagation

A very simple way of implementing a propagation-based error model is called sum propagation, which coincides with the conservative bound estimation presented in Section II. Here, each node is considered individually and, hence, the graph can be trivially partitioned so that each node corresponds to a subgraph.

Labels are positive integers, the weights, and a label $\ell \in L$ represents a bound on the absolute error, *i.e.* $\max_{\boldsymbol{x}} |e_{\boldsymbol{x}}| \leq \ell$.

In sum propagation, the propagation function p ignores the functionality implemented by a given node and assigns to it the sum of all its children's labels. This approach is used, for instance, in [2], and an example of a graph labelled through sum propagation is given in Figure 8.

Explicitly,

$$p_j(\ell, g) := \sum_{i=1}^{t} \ell_i \tag{8}$$

for all inputs $1 \leq j \leq s$ of the node under consideration. Note that g and j do not appear on the right-hand side of

the definition, leading to a fast but imprecise analysis. Since propagation is a linear operation, Eq. (8) can be expressed in matrix form:

$$p(\ell, g) := 1_s 1_t^T \ell$$

where 1_k notes a vector of ones of length k.

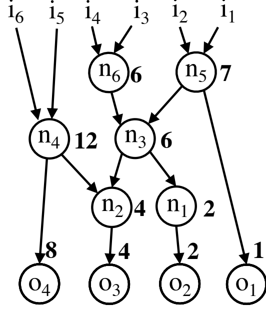

Fig. 8: Sum labelling.

B. P&P-Monotonicity

A more precise analysis can be obtained through a combination of larger subgraphs and a study of the functionality of each subgraph. This is done in P&P-Monotonicity (PP-Mon) [5], which is another example of prior work that fits into the presented framework. We will now describe this algorithm using the notation introduced above, and we will add some new insights on how monotonicity of subgraphs is exploited to improve weights accuracy.

In PP-Mon, labels are pairs of a numerical weight and a symbolic tag, $L = \mathbb{Z} \times \{S, NS, NM\}$, where weights are integers $w \in \mathbb{Z}$ and tags are associated attributes meaning respectively *strict monotonic*, *non-strict monotonic*, and *non-monotonic*: indeed, in PP-Mon, the monotonicity of the primary output w.r.t. each node is taken into account when propagating weights through subgraphs.

1) Propagation mechanism: Propagation is performed looking at the subgraph truth-table, which implements a Boolean function

$$g : \mathbb{B}^s \rightarrow \mathbb{B}^t \tag{9}$$

where s is the number of inputs of the truth table, and t the number of outputs. We, then, define function

$$g_{n,v} : \mathbb{B}^{s-1} \rightarrow \mathbb{B}^t \tag{10}$$

to be

$$g^{\boldsymbol{x}}_{n,v} = g(x_1, ..., x_{n-1}, v, x_{n+1}, ..., x_s). \tag{11}$$

Equation (11) represents the output of truth table g when its n-th input is forced to a constant value v. Note that it has one fewer input than g. We call $\boldsymbol{y} \in \mathbb{B}^{s-1}$ an input combination of function (11).

To derive the weight of bit n, the difference

$$|f(g^{\boldsymbol{y}}_{n,1}) - f(g^{\boldsymbol{y}}_{n,0})| \tag{12}$$

is computed for all $\boldsymbol{y} \in \mathbb{B}^{s-1}$, where f is the significance-weighting function (1).

The final value of w_n is, then:

$$w_n = \max_{\boldsymbol{y}}(|f(g^{\boldsymbol{y}}_{n,1}) - f(g^{\boldsymbol{y}}_{n,0})|) \tag{13}$$

Figure 9 illustrates an example of weight propagation, where a subgraph of two inputs (a, b), and two outputs (c, d) is depicted; weights of the two outputs are w_c and w_d.

In particular, Figure 9b illustrates how w_b is obtained: a pair-wise comparison is performed between input tuples that differ only for the value of bit b. In this example, $y \in \{y_0, y_1\}$, where $y_0 = [0]$ and $y_1 = [1]$. For $y = y_1$, $g^1_{b,0} = g(1, 0) = [0\ 1]$ and $g^1_{b,1} = g(1, 1) = [1\ 0]$; hence the difference computed in equation (12) is equal to $|w_c - w_d|$, which we will assume greater than w_c for the sake of explanation. Here is where tags on monotonicity come into play: if the integer value of the primary output vector $f(\boldsymbol{o})$ is strictly monotonically increasing with subgraph output bits c and d, these bits are tagged as S (for *strict monotonic*). We can safely set w_b to the *difference* of output bits weights because the strict monotonicity of $f(\boldsymbol{o})$ guarantees that errors at the subgraph output will propagate in the same way (*i.e.*, with the same polarity) to the graph primary outputs. Otherwise, if bits are tagged as NS or NM, it will not be possible to simply subtract the weights, as will be explained in detail in the following sections.

Weight w_b is then derived according to the subgraph output monotonicity, as follows:

$$p(\ell, g)_n := \max_{\boldsymbol{y} \in \mathbb{B}^{s-1}} |\ell^T \triangle (g^{\boldsymbol{y}}_{n,1} - g^{\boldsymbol{y}}_{n,0})| \tag{14}$$

where ℓ is the vector containing all labels of the subgraph output, and the usual scalar product is redefined with $\triangle$, which applies a different operation to each i-th partial product according to the corresponding bit monotonicity tag:

$$\triangle = \begin{cases} +\ell_i(g^{\boldsymbol{y}}_{n,1} - g^{\boldsymbol{y}}_{n,0})_i, & if\ output\ i\ is\ S \\ +\ell_i|(g^{\boldsymbol{y}}_{n,1} - g^{\boldsymbol{y}}_{n,0})_i|, & if\ output\ i\ is\ NM \end{cases}$$

while if output i is NS, the signed partial product $\ell_i(g^{\boldsymbol{y}}_{n,1} - g^{\boldsymbol{y}}_{n,0})_i$ is temporarily held aside, to check whether it increments the final result or not. The reason for this distinction will be clarified in point 3, below.

The same process is repeated for input a_1 and, since input labels are linear combinations of output labels, the results can be stored in a *propagation matrix* $M(|I_s|,\ |O_s|)$, where the i-th row contains the coefficients (0, 1 or -1) of the linear combination for input i:

$$p(\ell, g) = M(g)\ell$$

Propagation matrices are derived for each subgraph, when Equation (14) is applied to propagate subgraph output weights to their inputs, hence obtaining the matrices coefficients.

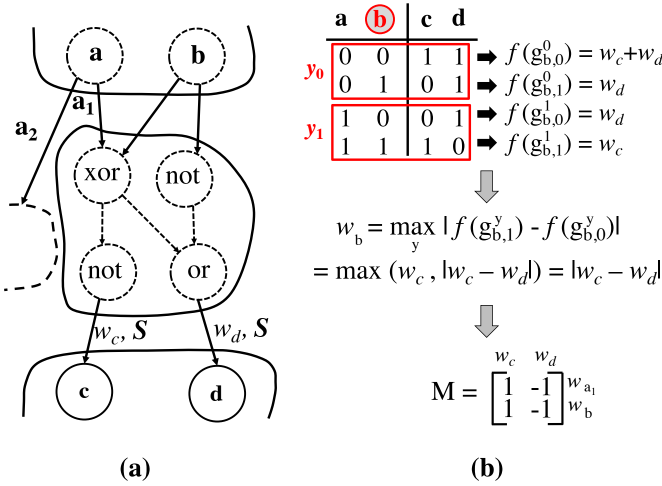

Fig. 9: Propagation matrix derivation for an example subgraph. The figure reports its truth table, and differences are computed for w_b. The process is repeated twice (once for each distinguishable input) to obtain the complete matrix.

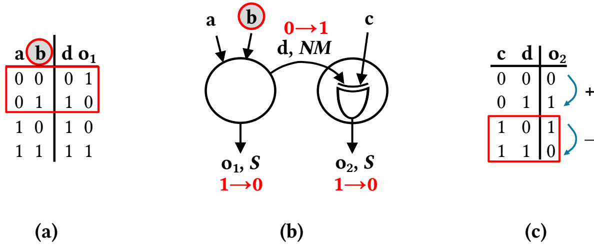

Fig. 10: An example of reverted propagation in non-monotonic output bits. A two-subgraph partition (b), with the truth table of the left subgraph (a) and that of the right one (c).

2) Non-monotonic outputs: If, on the contrary, $f(\boldsymbol{o})$ is not monotonically increasing with the subgraph output bits, the direction of subgraphs bit variations could be reverted in lower computations and, hence, we are forced to set the input weight to the *sum* of all flipped output bit weights, always taking the absolute value of the difference $(g^{\boldsymbol{y}}_{n,1} - g^{\boldsymbol{y}}_{n,0})_i$ in Eq. (14).

Figure 10 illustrates an example of this behaviour: in Fig. 10b we can observe a two-subgraph partition with primary outputs o_1 and o_2. These are by definition strictly monotonic (S): if their value increase, the primary output value $f(\boldsymbol{o})$ increases as well. When computing weight w_b, we observe from the truth table of the left subgraph (Fig. 10a) that the first comparison, enclosed in the red rectangle, would give again $|w_d - w_{o_1}|$. However, d is input to the right subgraph, whose truth table is depicted in Fig. 10c: here it can be seen that for an increase of bit d, the primary output can either increase or decrease, as highlighted by the red rectangle. In this case, the final effect on the primary output would be $|-w_{o_2} - w_{o_1}|$ and, therefore, w_b should be set to $w_d + w_{o_1}$. Indeed, bit d is tagged as non monotonic (NM).

Note that a single non-monotonic subgraph in the path to the primary outputs is sufficient for potential error underestimation; therefore, information on non-monotonicity must be retained for upper subgraphs and, hence, each output bit is

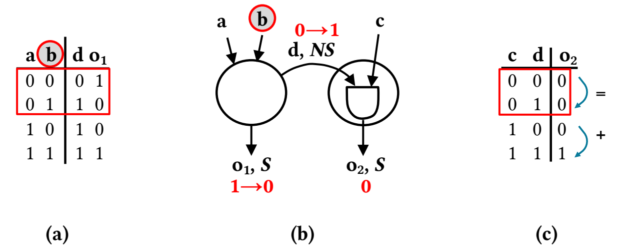

Fig. 11: An example of propagation in non-strict monotonic output bits.

tagged with information on monotonicity (*strict monotonic*, *non-strict monotonic*, and *non-monotonic*).

3) Non-strict monotonic outputs: It is also important to distinguish between strict and non-strict monotonicity. Figure 11 illustrates a non-strict monotonic bit d: indeed, in Fig. 11c it can be seen that if bit d increases, the primary output o_2 can either increase or remain constant. Therefore, there can be two possible outcomes on the primary outputs, depending on the value of bit c: $|-w_{o_1}|$ or $|w_d - w_{o_1}|$, and we will have to take the maximum between these two quantities. Hence, in case of multiple output bitflips, all partial products left aside in $\triangle$ of Eq. (14) will be either added or excluded from the final result, so to maximize it.

We provide below an example of a subgraph with seven outputs and their corresponding tags, where the difference $g_{n,1} - g_{n,0}$ is computed as in Eq. (14) to discover the sign of each bitflip.

$1,S$	$2,S$	$3,NM$	$4,NM$	$5,NS$	$6,NS$	$7,NS$		
0	1	0	1	1	1	0	=	$g_{n,0}$
1	0	1	0	0	0	1	=	$g_{n,1}$
1	-1	1	-1	-1	-1	1	=	$g_{n,1} - g_{n,0}$

Now, if output tags were all S, the product $\ell^T(g_{n,1} - g_{n,0})$ would give $\ell_1 - \ell_2 + \ell_3 - \ell_4 - \ell_5 - \ell_6 + \ell_7$. For outputs with an NM tag, instead, we need to take the *absolute value* of the corresponding bit in $g_{n,1} - g_{n,0}$, hence obtaining $\ell_1 - \ell_2 + \ell_3 + \boldsymbol{\ell_4} - \ell_5 - \ell_6 + \ell_7$. For outputs with an NS tag, since the bitflip effect can manifest or not (it could be zero), we'll need to check which of these cases maximises the final sum. In this example, we have three output bits with an NS tag, two of negative direction (bits 5 and 6) and one positive (bit 7). Therefore, the two most conservative cases are those where all negative bitflips manifest, while all positives do not, and *viceversa*:

$$\max(|\ell_1 - \ell_2 + \ell_3 + \ell_4 \boldsymbol{- \ell_5 - \ell_6}|, \ \ |\ell_1 - \ell_2 + \ell_3 + \ell_4 \boldsymbol{+ \ell_7}|)$$

Note that, if $(g_{n,1} - g_{n,0})_7$ were 0, hence output bit 7 did not flip, the first term of the comparison above would be the same, while the second would be $|\ell_1 - \ell_2 + \ell_3 + \ell_4|$, accounting for the case where negative bitflips of outputs 5 and 6 do not manifest.

Node tags are computed over the graph primary outputs, by verifying whether these are strictly monotonically increasing,

non-strictly increasing or non-monotonic with respect to each subgraph input. However, unless all subgraph outputs are primary outputs, this property cannot be directly observed. Therefore, the subgraph truth table along with the local output tags are used to propagate information on monotonicity: if, for example, the local output is monotonically increasing w.r.t. a given input, but one of the local output bits is tagged as NM, then the input will also be tagged as NM, since we cannot guarantee that an increase in the local output will reflect in an increase on the global output. Note also that, if a node has children in different subgraphs, it will automatically be tagged as NM, since there is no information on the combined effect of its children monotonicity.

It is interesting to remark that monotonicity is strictly connected with the choice of the error metric employed to measure the discrepancy between exact and approximate output. For instance, if Hamming distance was employed instead of numerical distance, all bits would be tagged as NM, since the Hamming distance between the exact and the approximate output would always increase, no matter what direction a given bitflip takes, and all primary output weights would be equal to 1.

4) Final error model derivation: The missing step at this point is to propagate weights from the *inputs* of generic subgraphs to the *outputs* of their *parent* subgraph(s). For a generic subgraph output, either all its children belong to the same subgraph (as for node b of Figure 9), or children nodes are distributed in different subgraphs (as for node a). Derivation for the first case is trivial: w_b is the one computed through the propagation matrix M. However, if a node has children belonging to different subgraphs, its weight must conservatively be computed as the sum of its children nodes weights ($w_{a_1} + w_{a_2}$ for w_a in the example), since the algorithm cannot resort to a single truth table to compute a less conservative weight.

Once external edges and subgraph outputs are labelled, each subgraph is populated with internal node weights. In this phase, exhaustive simulation is employed, as in [3], but applied to *each subgraph separately*, where the crucial difference is that the number of inputs of each subgraph is much smaller than that of the whole circuit $|I|$, hence ensuring computational tractability. Note that it is not necessary to compute tags for internal nodes, since their weights are never used to compute upper-level weights.

VII. P&P-Intervals

While PP-Mon presents a valid compromise between accuracy and scalability, the study of the primary output monotonicity w.r.t. subgraph output nodes introduces computational overhead.

In P&P-Intervals (PP-Int), labels $L = \mathbb{R} \times \mathbb{R}$ are intervals of values that enclose the error entailed on the output by the removal of node n_i:

$$I_i = [l_i, u_i] \tag{15}$$

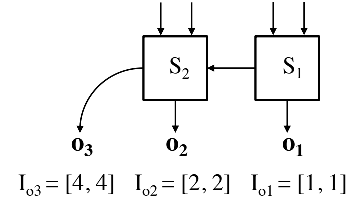

Fig. 12: A simple example of a 2-bit ripple-carry adder, with intervals assigned to its primary outputs.

where the two extremes l_i, u_i represent, respectively, a bound on the *minimum* and *maximum* error that can be observed on the circuit primary output when n_i flips from value 0 to value 1. The interval corresponding to a decrease of a node value from 1 to 0 is symmetric with the increasing one:

$$-I_i = I_{i,1\to 0} = [-u_i, -l_i] \tag{16}$$

We consider again $\boldsymbol{o}^{\boldsymbol{x}}_{i,0}$ and $\boldsymbol{o}^{\boldsymbol{x}}_{i,1}$, the two Boolean vectors of the primary outputs generated when node n_i is set to 1 and 0 respectively, for input x. The extremes of I_i are, then:

$$l_i = \min_{\boldsymbol{x}}(f(\boldsymbol{o}^{\boldsymbol{x}}_{i,1}) - f(\boldsymbol{o}^{\boldsymbol{x}}_{i,0})) \tag{17}$$

$$u_i = \max_{\boldsymbol{x}}(f(\boldsymbol{o}^{\boldsymbol{x}}_{i,1}) - f(\boldsymbol{o}^{\boldsymbol{x}}_{i,0})) \tag{18}$$

Enclosing all possible error values in an interval presents a major advantage when it comes to the choice of approximation: indeed, not only the value of the maximum error is available, as in PP-Mon, but its whole range. Moreover, once the interval is at hand, deriving the maximum possible discrepancy from the exact output (the weight) is trivial:

$$w(n_i) = \max(|l_i|, |u_i|) \tag{19}$$

However, values of equations (17) and (18) are, in general, unknown. We aim to find *bounds* on such values, which will satisfy the following:

$$l_i \leq \min_{\boldsymbol{x}}(f(\boldsymbol{o}^{\boldsymbol{x}}_{i,1}) - f(\boldsymbol{o}^{\boldsymbol{x}}_{i,0})) \tag{20}$$

$$u_i \geq \max_{\boldsymbol{x}}(f(\boldsymbol{o}^{\boldsymbol{x}}_{i,1}) - f(\boldsymbol{o}^{\boldsymbol{x}}_{i,0})) \tag{21}$$

In other words, we aim to identify the tightest possible interval containing the actual one.

A. Primary output intervals

As mentioned above, function (1) assigns a value to the Boolean vector of primary outputs, the most natural choice for such function being bit-significance weighting in the case where the output represents a single binary word. Therefore, primary output intervals are assigned with both extremes equal to the output bit significance.

Figure 12 illustrates a simple example of a 2-bit adder with three primary outputs, o_3, o_2 and o_1. When these output bits flip from 0 to 1, the integer value of the output increases by, respectively, 4, 2 and 1. Therefore, primary output bits intervals are set to $I_{o_3} = [4, 4]$, $I_{o_2} = [2, 2]$ and $I_{o_1} = [1, 1]$.

If, instead, one was interested in estimating the Hamming distance from the exact result, all primary outputs would have equivalent weight and, hence, all primary output intervals would be set to $[1, 1]$.

B. Input interval derivation and propagation

Once primary output intervals are assigned, the partition graph is traversed towards the PIs to compute intervals for all external edges (*i.e.*, those linking two different subgraphs). Each input interval is computed by comparing truth table lines where the inspected input flips from 0 to 1, while all other input remain constant, exactly as in PP-Mon. In Figure 13a, input c is inspected and these lines are enclosed in red rectangles.

As illustrated in Section VI-B, we represent each truth table as a function

$$g : \mathbb{B}^s \rightarrow \mathbb{B}^t$$

where s is the number of inputs of the truth table, and t the number of outputs. We have defined function $g_{n,v}$ of Eq. (11) as the output of truth table g when its n-th input is forced to a constant value v.

Figures 13 and 14 illustrate interval propagation for input c in a generic subgraph (fig. 13b). Here, $\boldsymbol{y} \in \{y_0, \ldots, y_3\}$, where $y_0 = [0\ 0]$, $y_1 = [0\ 1]$, *etc.*, representing all possible combination of inputs a and b. Each value of $\boldsymbol{y}$ identifies a pair of truth table lines employed to compute a partial interval $I^{\boldsymbol{y}}$.

Figure 14 illustrates partial interval computations *per single value of* $\boldsymbol{y}$. For example, when $\boldsymbol{y} = y_0$, a and b are equal to 0, and equation (11) provides values

$$g^{\mathbf{0}}_{c,0} = g(0, 0, 0) = [0\ \ 0]$$
$$g^{\mathbf{0}}_{c,1} = g(0, 0, 1) = [0\ \ 1]$$

because we fix the value of input c to 0 and to 1.

$I^{\mathbf{0}}$ is computed by subtracting the weighted value of $g^{\mathbf{0}}_{c,0}$ from that of $g^{\mathbf{0}}_{c,1}$:

$$I^{\mathbf{0}} = [I_d\ I_e](g^{\mathbf{0}}_{c,1} - g^{\mathbf{0}}_{c,0}) = I_e = [0, 2]$$

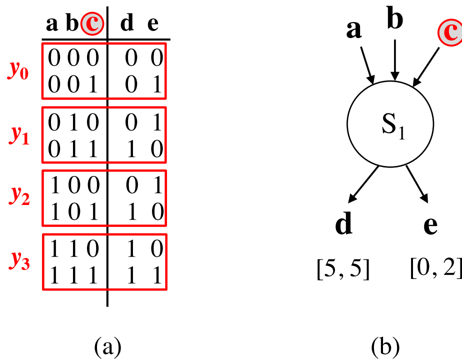

Fig. 13: A generic subgraph (b) with its truth table (a), which is used to compute intervals for its inputs.

	a b c	d e	
y_0	0 0 0	0 0	$I_e = [0, 2] = I^0$
	0 0 1	0 1	
y_1	0 1 0	0 1	$I_d - I_e = [5, 5] + [-2, 0] = [3, 5] = I^1$
	0 1 1	1 0	
y_2	1 0 0	0 1	$I_d - I_e = [5, 5] + [-2, 0] = [3, 5] = I^2$
	1 0 1	1 0	
y_3	1 1 0	1 0	$I_e = [0, 2] = I^3$
	1 1 1	1 1	

Fig. 14: Detailed computation of intervals for bit c of Figure 13a.

For interval $I^{\mathbf{1}}$, instead, we have $I^{\mathbf{1}} = [I_d\ I_e](g^{\mathbf{1}}_{c,1} - g^{\mathbf{1}}_{c,0}) = I_d - I_e = [5, 5] - [0, 2] = [3, 5]$. The final interval for bit c is, then, the *smallest* interval containing all 2^{s-1} partial intervals $I^{\boldsymbol{y}}$: $[0, 5]$, in this simple example.

Therefore, the propagation function for input n is:

$$p(\ell, g)_n := \uplus_{\boldsymbol{y} \in \mathbb{B}^{s-1}} \ell^T (g^{\boldsymbol{y}}_{n,1} - g^{\boldsymbol{y}}_{n,0}) \quad (22)$$

where $\uplus$ indicates interval union, denoting the smallest interval containing all argument intervals. Equation (22) can also be expressed in matrix form, since the input intervals extremes are linear combination of the output interval extremes:

$$p(\ell, g) = N(g)\ell$$

where N(g) contains, as in PP-Mon, coefficients 1, 0 or -1; while ℓ is a vector of the form $[l_1\ u_1\ \ldots\ l_t\ u_t]$ with all output labels extremes.

After all input intervals of a given subgraph are computed, the process is repeated for upper level subgraphs, and interval values are propagated to the graph primary inputs. If a node has children belonging to different subgraphs, its interval will be set to the sum of its children's intervals.

Note that we would always employ the sum of children's intervals also when estimating the maximum Hamming distance from the exact result, as opposed to the arithmetic distance. Indeed, we must remember that the Hamming distance increases independently from the bitflip direction.

Finally, we resort to exhaustive simulation on each subgraph to compute internal node intervals, similarly to PP-Mon.

The key characteristics of this method, as well as those described in the previous sections, are summarized in Table I for rapid comparison.

C. Strengths of PP-Int

The strength of PP-Int over PP-Mon resides in the choice of intervals for label representation, which allows faster – and more elegant – propagation by ignoring information on monotonicity, and provides lower bounds for errors. In fact, there is no need to consider how a given bitflip will impact on the primary output (either positively or negatively), because this information is *intrinsically* contained in the error interval. Moreover, since intervals purely track the error propagation

TABLE I: Summary of the three algorithms subsumed by our formal framework.

method	**ref.**	**labels**	$p(\ell, g)$	**matrix form**
sum	[2]	$L = \mathbb{Z}$	$\sum_{i=1}^{t} \ell_i$	$1_s 1_t^T \ell$
PP-Mon	[5]	$L = \mathbb{Z} \times \{S, NS, NM\}$	$\max_{\boldsymbol{y} \in \mathbb{B}^{s-1}} \lvert \ell^T \triangle (g_{n,1}^{\boldsymbol{y}} - g_{n,0}^{\boldsymbol{y}}) \rvert$	$M(g)\ell$
PP-Int	-	$L = \mathbb{R} \times \mathbb{R}$	$\biguplus_{\boldsymbol{y} \in \mathbb{B}^{s-1}} \ell^T (g_{n,1}^{\boldsymbol{y}} - g_{n,0}^{\boldsymbol{y}})$	$N(g)\ell$

TABLE II: Characteristics of benchmarks employed in [5].

benchmark	**I/O**	**gates**	**delay (ns)**	**description**
ADDER8	16/9	115	0.5	8-bit adder
BUTTFLY	32/33	485	2.0	simple butterfly structure
ABS_DIFF	16/9	245	0.1	8-bit absolute difference
ADDER32	64/33	475	2.0	32-bit adder
MULT8	16/16	685	2.0	8-bit unsigned multiplier
BIN_SQ	16/18	946	5.0	8-bit binomial squared

TABLE III: Characteristics of new benchmarks.

benchmark	**I/O**	**gates**	**delay (ns)**	**description**
O_ADDER4	18/10	137	1.0	4-bit online adder
O_ADDER8	34/18	140	2.0	8-bit online adder
O_ADDER32	130/66	549	5.0	32-bit online adder
ADDER48_2	96/49	833	2.0	48-bit adder, 2 ns
ADDER48_5	96/49	806	5.0	48-bit adder, 5 ns
DIST	64/32	5344	50.0	16-bit euclidean distance
MADD	24/16	840	5.0	8-bit multiply-add unit
MULT16_1	32/32	3811	1.0	16-bit unsigned multiplier, 1 ns
MULT16_5	32/32	2572	5.0	16-bit unsigned multiplier, 5 ns
SAD_10	80/16	2893	10.0	16-bit sum of absolute difference, 10 ns
SAD_20	80/16	2385	20.0	16-bit sum of absolute difference, 20 ns
SAD_50	80/16	1998	50.0	16-bit sum of absolute difference, 50 ns
ADDER32_1	64/33	487	1.0	32-bit adder, 1 ns
ADDER32_1.5	64/33	474	1.5	32-bit adder, 1.5 ns
ADDER32_5	64/33	537	5.0	32-bit adder, 5 ns

across the graph and do not need to distinguish between *sum*, *difference* or *maximum* (as PP-Mon does), the corresponding weights can be less conservative. This happens when non-symmetric intervals are summed together: as a simple example, consider intervals [-3, 1] and [5, 5]. Their sum gives [2, 6], which corresponds to a weight of 6, while in PP-Mon the first interval would have been associated to a non-monotonic output of weight 3, giving a weight of 8 as final result.

However, as can be seen in our experiments in the next section, these two approaches can sometimes lead to similar weight values. Indeed, they both aim at providing bounds on the error range and, while PP-Int is often less conservative – and this is a significant improvement over PP-Mon – we expect comparable results. In some benchmarks though, such as online adders, the difference in weights is much more pronounced, as illustrated in Section VIII.

The principal strength of this method resides in its simplicity and in the enriched information provided for error-modeling.

VIII. Experimental evaluation

To assess the performance of the presented methodology, we have implemented the three error propagation techniques over a wide set of benchmark circuits specified in VHDL. We have first considered the same benchmarks employed in [5], whose characteristics are reported in Table II, then we have expanded this benchmark set by adding the circuits listed in Table III. All circuits were synthesized with Synopsys Design Compiler, targeting a 40nm technology library, while exhaustive simulation and sub-graph simulation was performed with SIS [26]. For both PP-Mon and PP-Int we have employed the partitioning strategy described in [5], setting the maximum number of inputs per subgraph to 10.

A. PP-Int interval values

As described in Section VII, PP-Int assigns to each circuit gate a label in the form of eq. (15), with a lower bound l_i and an upper bound u_i. To derive the corresponding weight, we take the maximum absolute value of the two, as in eq. (19). Figure 15 reports these three values for each gate of the two benchmarks ADDER8 and ABS_DIFF, where gates are sorted by increasing weight value. It can be observed that, even for large weight values, ADDER8 presents narrow intervals, which seldom span from $-w(n_i)$ to $w(n_i)$. On the contrary, ABS_DIFF shows more variability, with wide symmetric intervals for many gates. Indeed, as described in the following section and illustrated in Figure 16, ADDER8 weights coincide with the simulated ones, while ABS_DIFF weights are, generally, more distant.

B. Comparison with benchmarks used in [5]

Figure 16 compares weights obtained by PP-Int, derived from the corresponding interval, against the state-of-the-art

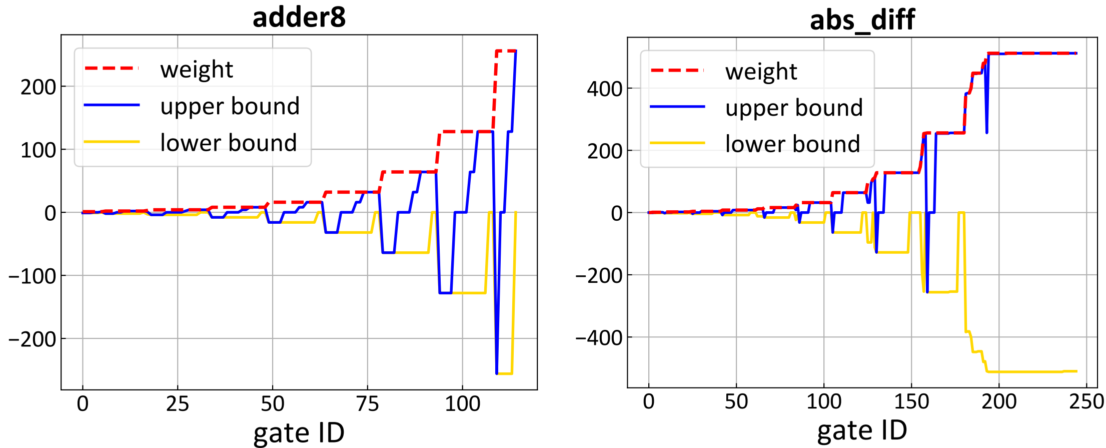

Fig. 15: Lower and upper bounds found by PP-Int, and the corresponding weight, for ADDER8 (left) and ABS_DIFF (right).

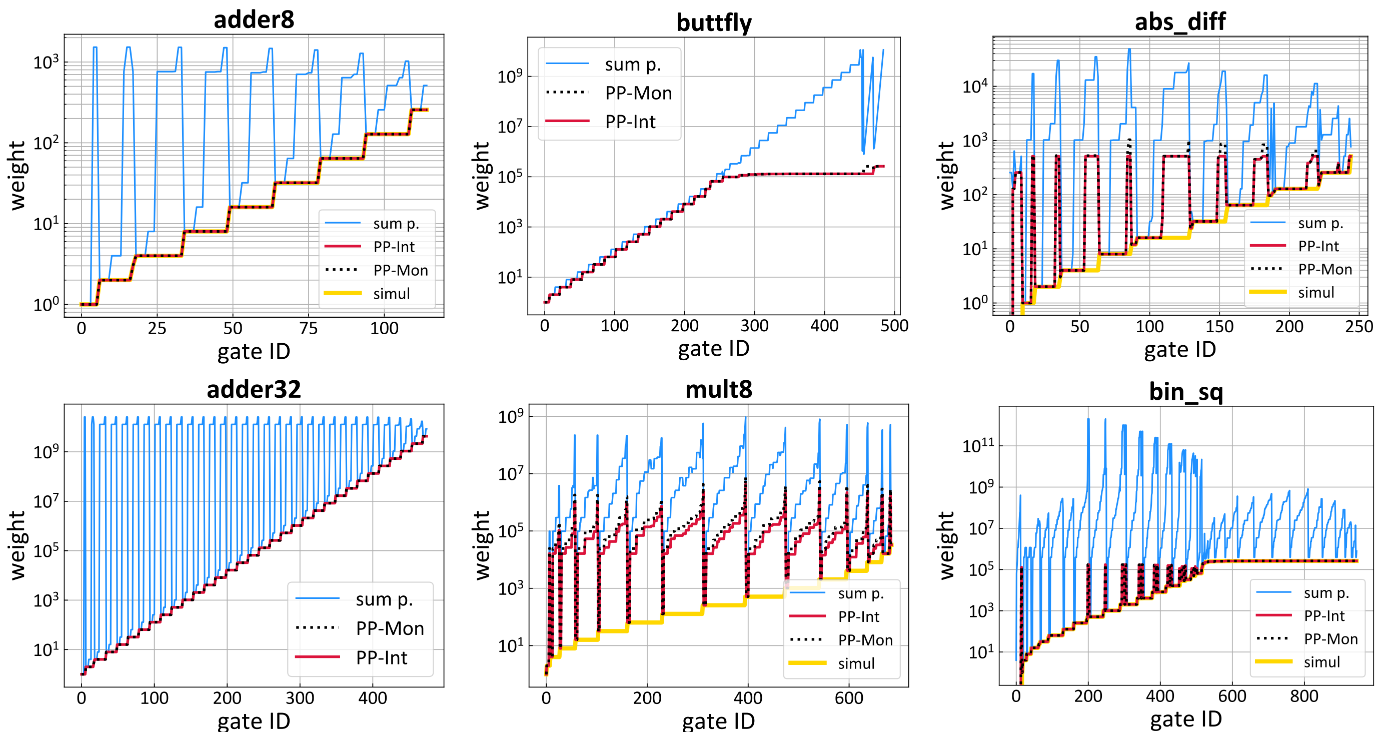

Fig. 16: Comparison of weights obtained through sum propagation, PP-Mon and PP-Int for the benchmarks employed in [5]. The weight of each gate is reported, and gates sorted by increasing (simulated or PP-Int) value.

strategies sum propagation and PP-Mon. For circuits with up to 16 inputs, also exhaustive simulation has been performed. For each circuit, the weight of each gate is reported, and gates are disposed on the x-axis by increasing simulated weight, when available, and otherwise by increasing PP-Int weight. We can observe that PP-Mon and PP-Int weights are almost always identical, except for MULT8 benchmark and for a very small set of gates in BUTTFLY. As introduced in Section VII-C, the main advantage of PP-Int does not reside in strongly improved weight values, but rather in the simplicity, and therefore effectiveness, of the propagation mechanism. Moreover, weights confirm to be very close (or even equal) to the simulated ones, and still orders of magnitude lower than those obtained through sum propagation (for example, seven orders of magnitude lower for gate with ID=200 in BIN_SQ).

C. Expanded benchmark set

We have expanded [5] benchmark set with those of Table III and, in particular, we have introduced larger benchmarks to assess the efficiency of the proposed approach. Figure 17

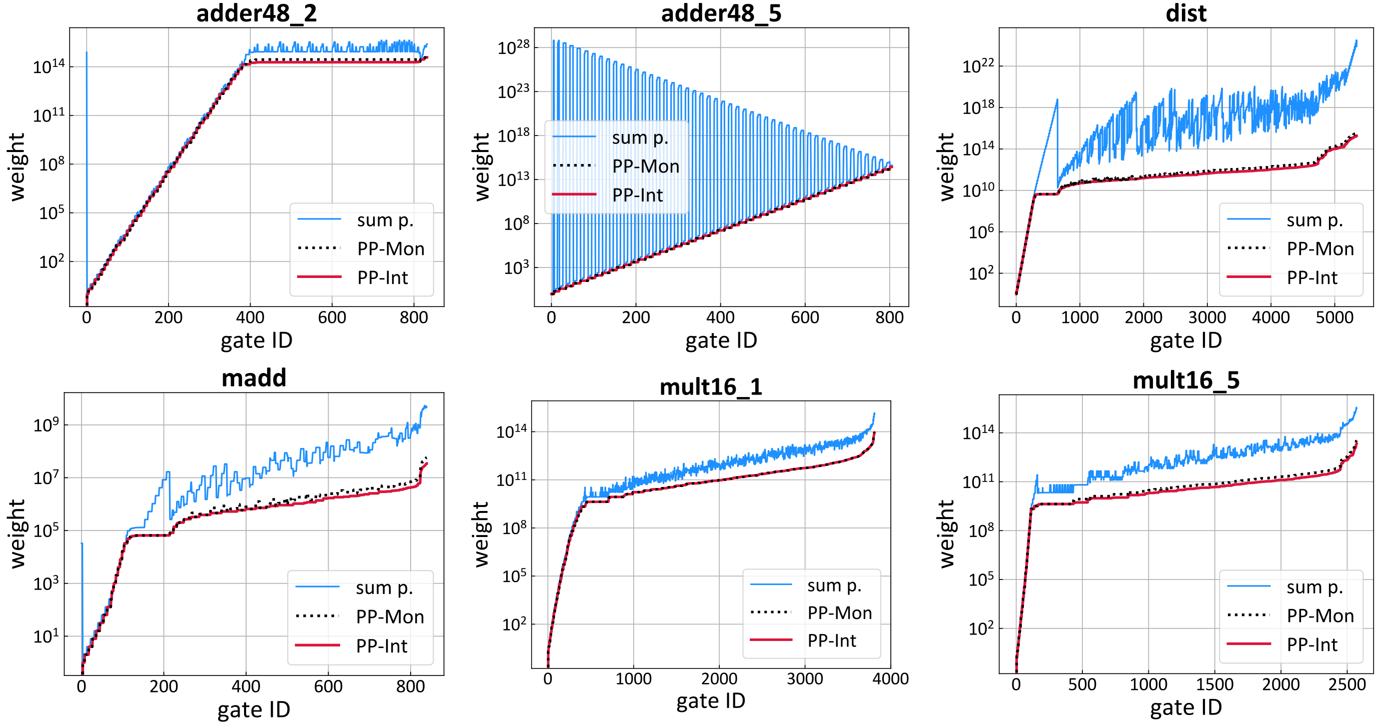

Fig. 17: Comparison of weights obtained through sum propagation, PP-Mon [5] and PP-Int for adders and euclidean distance (top-row), multiply-add and a multiplier (bottom row).

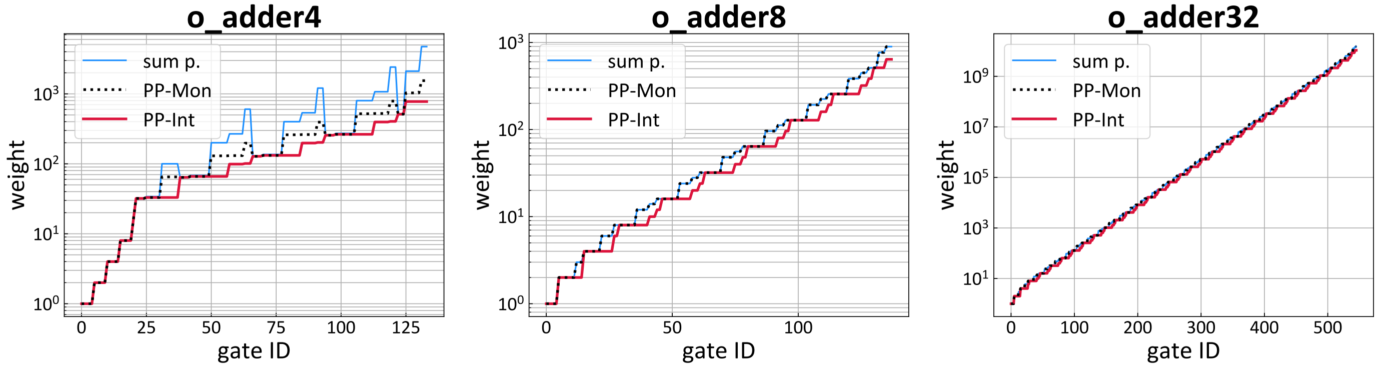

Fig. 18: Comparison of weights obtained through sum propagation, PP-Mon [5] and PP-Int for online adders.

reports six large benchmarks, whose average gate count is $\approx$ 2400 (with a maximum of 5344 for the DIST circuit).

In the top row of Figure 17 we can observe two 48-bit adders synthesised with different constraints on their critical path. In particular, ADDER48_2 has critical path of 2 ns, while ADDER48_5 of 5 ns. PP-Int and PP-Mon retrieve much lower weights than sum propagation for the slower adder (ADDER48_5), while for the faster adder (ADDER48_2) the difference is less pronounced (although still of one order of magnitude): this is due to the more complex structure of the faster adder, where it is harder to isolate monotonic subgraphs, or subgraphs leading to narrow intervals. A similar effect can be seen for the two 16-bit multipliers MULT16_1 and MULT16_5, and will be further illustrated in section VIII-E.

In the two remaining benchmarks MADD and DIST, the difference between weights obtained by sum propagation and those obtained by PP-Int and PP-Mon is again of several orders of magnitude (up to 9 in DIST).

In general, for four out of six benchmarks, PP-Int weights are slightly lower than PP-Mon ones, confirming the validity of the proposed approach.

D. *Online adders*

Online adders are special adder architectures which employ redundancy in data representation, allowing less-significant digits to correct errors introduced in those of higher signif-

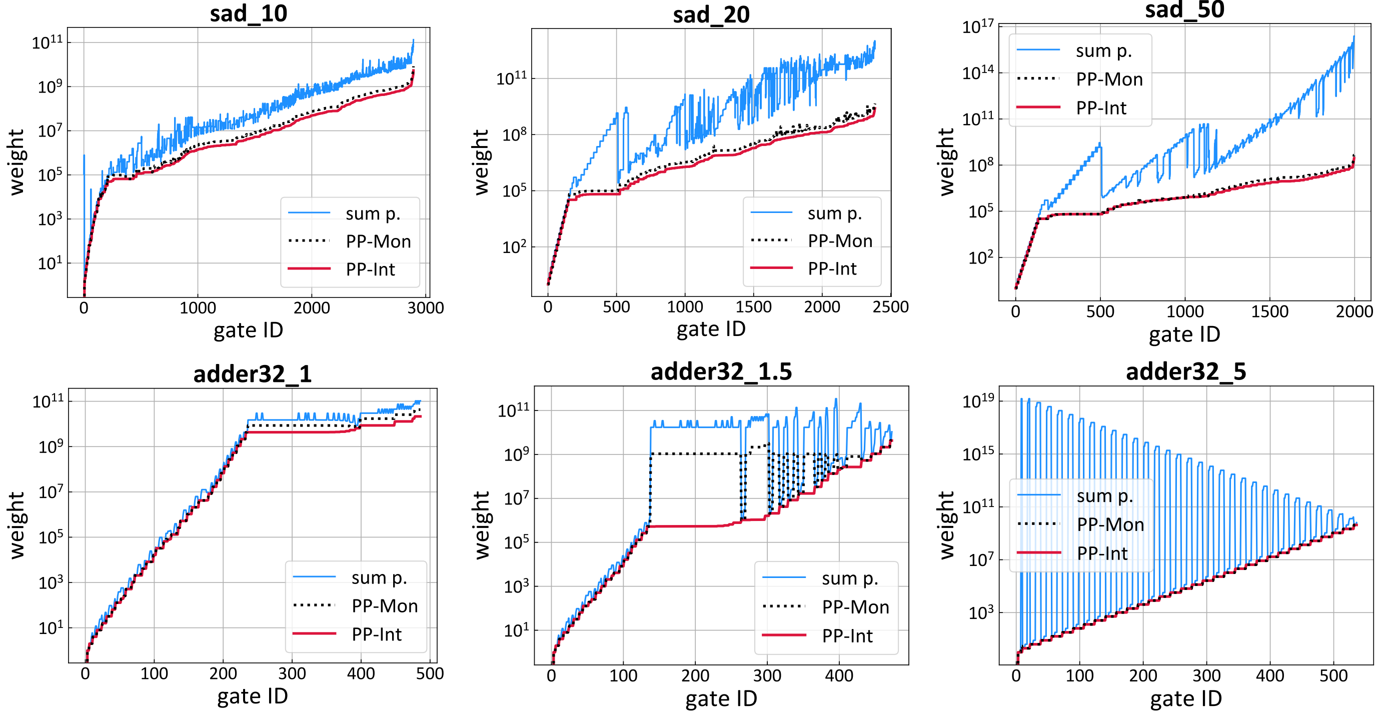

Fig. 19: Comparison of weights obtained through sum propagation, PP-Mon [5] and PP-Int for SAD with different delay constraints (top row) and 32-bit adders (bottom row).

icance, and functioning in MSD-first fashion [27], [28]. In these adders, each input bit x_i corresponds to a pair of bits, x_i^+ and x_i^-, selected such that $x_i = x_i^+ - x_i^-$ [27]. This is why O_ADDER8 (the 8-bit online adder) has 34 inputs (four 8-bit inputs, plus a 2-bit carry-in) and 18 outputs (two 8-bit outputs, and a 2-bit carry-out). Their structure looks similar to that of a ripple-carry adder, but there is no carry chain running through the graph, leading to a partition that isolates full-adder-like blocks, but with different subgraph functionalities. Figure 18 reports weights obtained by the three techniques for 4, 8 and 32-bit online adders. For the second and third circuit, sum propagation and PP-Mon weights even coincide, while PP-Int is able to systematically retrieve lower weights in all cases. This demonstrates once again that PP-Int can improve weights, especially when non-monotonic subgraphs hamper PP-Mon.

E. Faster vs slower circuits

Finally, we have further studied weights accuracy for circuits with the same functionality but with different topological structure, imposed by different delay constraint at synthesis time.

Figure 19 showcases weights retrieved for three 16-bit SAD circuits (top row) and three 32-bit adders (bottom row). Their delay constraint decreases left to right, so that leftmost circuits are the fastest and, consequently, the largest. The results confirm the trend already seen in other benchmarks: for faster circuits, the difference between weights obtained by sum propagation and by PP-Int and PP-Mon is visible and can be of two orders of magnitude (see gate with ID 500 in SAD_10), but it increases remarkably in slower circuits (7 orders of magnitude for gate with ID 2000 in SAD_50, or even 15 orders of magnitude for gate with ID 100 in ADDER32_5).

Except from ADDER32_5, where PP-Int and PP-Mon weights coincide, in all other benchmarks PP-Int performs better than PP-Mon. In particular, we can see that weights are significantly improved for ADDER32_1.5, where for a large portion of gates the difference between PP-Int and PP-Mon reaches three orders of magnitude.

To sum up, we can conclude that the efficiency of PP-Int and PP-Mon can vary significantly according to the circuit structure, but these methods always perform better than sum propagation and, in many cases, PP-Int improves even considerably the accuracy of weights obtained through PP-Mon.

F. Effectiveness of error modeling for ALS

As motivated at the beginning of this paper, the importance of an accurate error-modeling phase is crucial for guiding ALS methods towards efficient solutions. To demonstrate this, we have compared the performance of approximate circuits obtained with the approach described in [2], called Gate Level Pruning (GLP), which iteratively removes one gate from the circuit and simulates the result, until the given error threshold is violated. Specifically, we have compared the Energy, Delay and Area Product (EDAP) of circuits obtained by GLP when guided by sum propagation, as in its original implementation, and when guided by our PP-Int. Figure 20 illustrates the obtained results: for the same amount of tolerated error, GLP

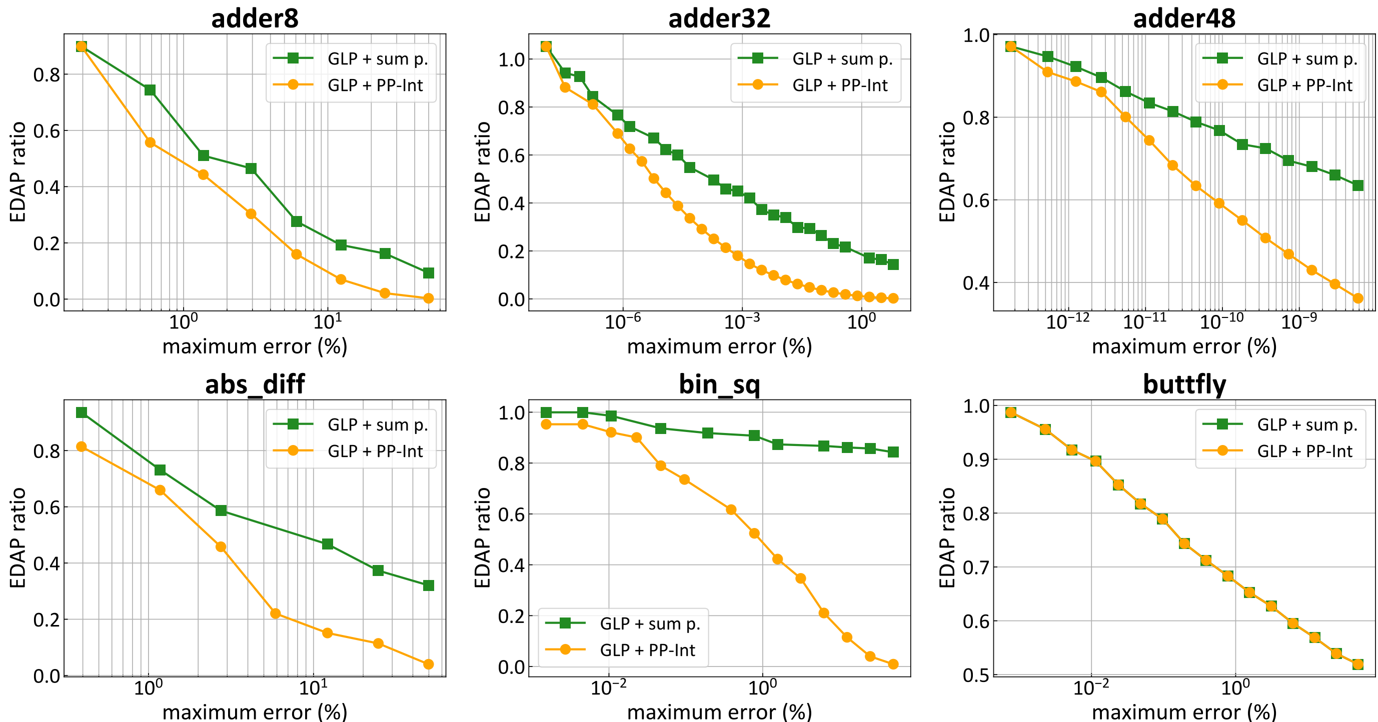

Fig. 20: Comparison of EDAP for circuits obtained with GLP [2] guided by sum propagation *vs* PP-Int. The error constraint is a measure of maximum error, expressed as a percentage of the maximum circuit output.

guided by PP-Int retrieves smaller, faster and less power-consuming circuits than those obtained when sum propagation is applied. For example, the EDAP is halved for a maximum tolerated error of 1% in BIN_SQ. These results further showcase the validity of the proposed approach in increasing the performance of ALS algorithms.

G. Runtime comparison for PP-Int and PP-Mon

As introduced in Section VII-C, the greatest advantage of PP-Int resides in the simplicity of label propagation. To assess the extent of such improvement, we have compared the label propagation runtime of PP-Mon and PP-Int over all benchmarks listed in Table II and III. These runtime values range from few seconds to several minutes, up to 14 minutes for BIN_SQ. Although BIN_SQ is not the largest benchmark in terms of number of gates, its subgraphs have a high number of inputs (up to 9), which explains the long runtime, since subgraph simulation is exponential in the number of the inputs. Indeed, runtime strongly depends on the partition structure found.

The results obtained confirm our insight: on average, PP-Int reduces the runtime by 11%. However, if we only consider those benchmarks for which propagation is slow, *i.e.*, it takes longer than a minute, the average runtime reduction amounts to 42%. We therefore conclude that, especially for complex partition structures with several propagation steps, PP-Int represents a significant improvement over PP-Mon.

TABLE IV: Influence of $\boldsymbol{T}$, the maximum number of inputs allowed per subgraph, on the resulting weight accuracy of BIN_SQ.

$\boldsymbol{T}$	10	8	5	4	3	2
number of sg.	8	117	123	126	136	206
avg. dist.	9.81e3	1.60e7	1.66e7	1.66e7	1.73e7	2.23e7
runtime (s)	849.8	63.2	37.7	36.1	34.2	27.2

H. Maximum number of inputs per subgraph

Finally, we have studied the effect of varying the maximum number of inputs T allowed per subgraph on the resulting weight accuracy. Less inputs per subgraph imply smaller subgraphs and, hence, more nodes with children belonging to different subgraphs, which will force the employment of conservative sum propagation. On the other hand, allowing too many inputs will result in infeasible simulation. We remind the reader that, for all the experiments presented in this section, threshold T was set to 10. Table IV reports these results for weights obtained by PP-Int on BIN_SQ: when T decreases, both the total number of subgraphs and the average distance from the exact weights increase, while the runtime decreases. Figure 21 showcases this effect, where T is indicated between brackets. While weights obtained by PP-Int with a threshold of 10 inputs per subgraph are close to the simulated ones, when the threshold is lowered to 5 they approach sum propagation.

IX. Conclusion

In this work we have enriched the state of the art in error modeling by introducing a formal framework for maximum

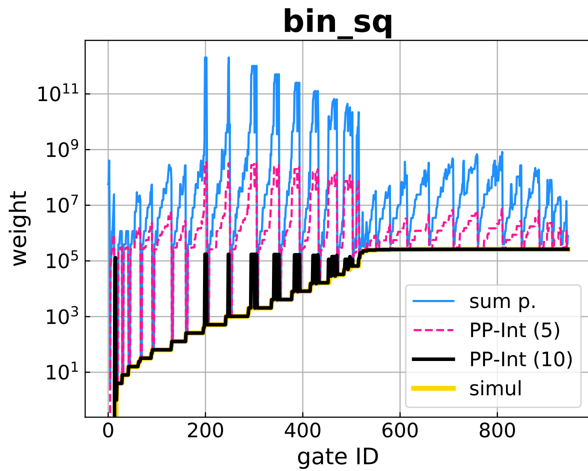

Fig. 21: Weights obtained by PP-Int on BIN_SQ with different values of maximum number of inputs per subgraph, respectively 5 (pink) and 10 (black).

error-propagation, Partition and Propagate, where the circuit is partitioned and the gates are labelled with values representing their influence on the final output. These labels are then propagated through the circuit via a propagation function. Several different error-modeling strategies originate from this framework, according to the choice of the partitioning algorithm and the propagation function. We have showcased how two prior-work methods fit in the described framework, and we have presented a third, new error-modeling strategy called P&P-Intervals. We have compared the three strategies over a wide set of benchmark circuits, outlining their different performance and demonstrating the effectiveness of the novel P&P-Intervals error-modeling method in estimating accurately gate-level errors.

References

[1] Q. Xu, T. Mytkowicz, and N. Kim, "Approximate computing: A survey," *IEEE Design and Test*, vol. 33, pp. 8–22, Jan. 2016.

[2] J. Schlachter, V. Camus, K. V. Palem, and C. Enz, "Design and applications of approximate circuits by gate-level pruning," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, pp. 1694–1702, Feb. 2017.

[3] I. Scarabottolo, G. Ansaloni, and L. Pozzi, "Circuit Carving: A methodology for the design of approximate hardware," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 545–550, Mar. 2018.

[4] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 1367–1372, Mar. 2013.

[5] I. Scarabottolo, G. Ansaloni, G. Constantinides, and L. Pozzi, "Partition and Propagate: an error derivation algorithm for the design of approximate circuits," in *Proceedings of the 56th Design Automation Conference*, pp. 1–6, June 2019.

[6] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan, "MACACO: Modeling and analysis of circuits for approximate computing," in *Proceedings of the International Conference on Computer Aided Design*, pp. 667–673, Nov. 2011.

[7] S. Su, Y. Wu, and W. Qian, "Efficient batch statistical error estimation for iterative multi-level approximate logic synthesis," in *Proceedings of the 55th Design Automation Conference*, pp. 54:1–54:6, June 2018.

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

[9] S. S. Basu, L. G. Duch, R. Braojos Lopez, G. Ansaloni, L. Pozzi, and D. Atienza Alonso, "An inexact ultra-low power bio-signal processing architecture with lightweight error recovery," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, Oct. 2017.

[10] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, *et al.*, "Razor: A low-power pipeline based on circuit-level timing speculation," in *MICRO 36: Proceedings of the 36th Annual International Symposium on Microarchitecture*, pp. 7–18, Dec. 2003.

[11] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu, "On reconfiguration-oriented approximate adder design and its application," in *Proceedings of the International Conference on Computer Aided Design*, pp. 48–54, Nov. 2013.

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

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

[14] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "SALSA: systematic logic synthesis of approximate circuits," in *Proceedings of the 49th Design Automation Conference*, pp. 796–801, June 2012.

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

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

[17] M. Soeken, D. Große, A. Chandrasekharan, and R. Drechsler, "BDD minimization for approximate computing," in *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 474–479, Jan. 2016.

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

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

[20] J. Castro-Godínez, S. Esser, M. Shafique, S. Pagani, and J. Henkel, "Compiler-Driven error analysis for designing approximate accelerators," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 1–6, Mar 2018.

[21] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "Axnn: energy-efficient neuromorphic systems using approximate computing," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 27–32, Aug. 2014.

[22] B. Moons, B. De Brabandere, L. Van Gool, and M. Verhelst, "Energy-efficient convnets through approximate computing," in *IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 1–8, Mar. 2016.

[23] A. Chandrasekharan, M. Soeken, D. Große, and R. Drechsler, "Approximation-aware rewriting of aigs for error tolerant applications," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, Nov 2016.

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

[25] Z. Zhang, Y. He, J. He, X. Yi, Q. Li, and B. Zhang, "Optimal slope ranking: An approximate computing approach for circuit pruning," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, May 2018.

[26] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," 1992.

[27] H. Li, J. J. Davis, J. Wickerson, and G. A. Constantinides, "architect: Arbitrary-precision hardware with digit elision for efficient iterative compute," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 516–529, 2020.

[28] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2003.

Ilaria Scarabottolo received the Ph.D. degree in Informatics at Università della Svizzera Italiana, Lugano, Switzerland, in 2020. In 2019, she was awarded the Swiss National Foundation grant DOC-Mobility, thanks to which she spent six months as visiting student at Imperial College London. She received her M.Sc. degree in Computer Engineering at Politecnico di Milano, Italy, in 2016. In parallel, she has obtained a second M.Sc degree at École Centrale Paris, France, in 2014. Her research interests include approximate computing, approximate logic synthesis for error-tolerant applications, embedded systems and low-power hardware design.

Giovanni Ansaloni is a researcher at the Embedded Systems Laboratory of EPFL (Lausanne, Switzerland). He previously worked as a Post-Doc at the University of Lugano (USI, Switzerland) between 2015 and 2020, and at EPFL between 2011 and 2015. He received a MS degree in electronic engineering from University of Ferrara (Italy) in 2003, an executive master in embedded systems design from the ALaRI institute (Switzerland) in 2005 and a PhD degree from USI in 2011. His research efforts focus on domain-specific and ultra-low-power architectures and algorithms for edge computing systems, including hardware and software optimization techniques.

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

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