Less is more: efficient hardware design through Approximate Logic Synthesis

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Ilaria Scarabottolo

under the supervision of
Prof. Laura Pozzi

October 2020

Dissertation Committee

Prof. George A. Constantinides	Imperial College London, United Kingdom
Prof. Jörg Henkel	Karlsruhe Institute of Technology, Germany
Prof. Akash Kumar	Technische Universität Dresden, Germany
Prof. Cesare Alippi	Università della Svizzera italiana, Switzerland
Prof. Antonio Carzaniga	Università della Svizzera italiana, Switzerland
Prof. Laura Pozzi	Università della Svizzera italiana, Switzerland

Dissertation accepted on 15 October 2020

Research Advisor	PhD Program Director
Prof. Laura Pozzi	**Prof. Walter Binder, Prof. Silvia Santini**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Ilaria Scarabottolo
Lugano, 15 October 2020

One never notices what has been done; one can only see what remains to be done.

Marie Curie

Abstract

As energy efficiency becomes a crucial concern in almost every kind of digital application, Approximate Computing gains popularity as a potential answer to this ever-growing energy quest.

Approximate Computing is a design paradigm particularly suited for error-resilient applications, where small losses in accuracy do not represent a significant reduction in the quality of the result. In these scenarios, energy consumption and resources employment (such as electric power, or circuit area) can be significantly improved at the expense of a slight reduction in output accuracy.

While Approximate Computing can be applied at different levels, my research focuses on the design of approximate hardware. In particular, my work explores Approximate Logic Synthesis, where the hardware functionality is automatically tuned to obtain more efficient counterparts, while always controlling the entailed error. Functional modifications include, among others, removal or substitution of gates and signals. A fundamental prerequisite for the application of these modifications is an *accurate error model* of the circuit under exam.

My Ph.D. research work has deeply concentrated on the derivation of accurate error models of a circuit. These can, in turn, guide Approximate Logic Synthesis algorithms to optimal solutions and avoid expensive, time-consuming simulations. A precise error model allows to fully explore the design space and, potentially, adjust the desired level of accuracy even at runtime. I have also contributed to the state of the art in ALS techniques by devising a circuit pruning algorithm that produces efficient approximate circuits for given error constraints.

The innovative aspect of my work is that it exploits circuit topology and graph partitioning to identify circuit portions that impact to a smaller extent on the final output. With this information, ALS algorithms can improve their efficiency by acting first on those less-influent portions. Indeed, this error characterisation proves to be very effective in guiding and modeling approximate synthesis.

Acknowledgements

This thesis is the result of several years of work, although technical research represents only a small fraction of it. These have been years of excitement, learning, self-questioning, doubts about the future, pride, and great personal development.

I cannot start but from thanking my research advisor Prof. Laura Pozzi, who has been much more than that throughout these years. Thank you Laura, for all your precious advice, your ever-present support and your motivation in pursuing my objectives. What I'd like to thank you most for, though, is the constant reminder you gave me on my professional and personal value, and on the importance of acknowledging it.

I often joke about the fact that I'd have never earned a Ph.D. without Dr. Lorenzo Ferretti, and I wonder how far this is from reality. Thank you Lorenzo for always being willing to help me with my endless questions, and for all the fun time we have spent together.

I will never thank my family enough for the unconditional support they have always given me since my earliest days, and which of course did not fade for a second during these past few years.

Finally, I'd like to thank all my closest friends. Thank you for all the joy and laughter, for reminding me that no problem is as tough as it seems, and for celebrating this great milestone like no other.

Contents

Figures

Tables

Chapter 1

What is Approximate Computing?

Technology has now manifestly become a pervasive aspect of social, industrial and economical life. The constant growth of the amount of data exchanged, the energy required to maintain IT services and the astonishing diffusion of mobile devices require a fast and reactive performance improvement on one side, and highlight challenging issues regarding energy supply on the other. Indeed, energy efficiency should nowadays be regarded as a most crucial concern in every kind of application.

At the same time, several energy-hungry applications are gaining so much popularity that their role is becoming fundamental in shaping our society, and their environmental impact cannot be ignored. Some examples are image processing, machine learning, robotics, wireless sensor networks and, in general, digital signal processing. In traditional computing, exactness of result has always been a cornerstone, but this exactness is not needed anymore in most of these applications; nonetheless, today these are still implemented on platforms and accelerators born under the traditional paradigm of full precision and reliability.

Approximate Computing (AC) represents a valid instrument to answer the constantly growing need for high performance at low energy cost of error-resilient applications, since it can lead to substantial energy savings in exchange for small losses in the output quality. As a simple example, in image processing a small quality loss in digital pictures cannot even be perceived by human eyes. The key idea under Approximate Computing is to relax requirements on exactness, in favour of improved energy efficiency, which in turn manifests in reduced hardware area and resource employment, or faster execution time.

Thanks to the growing diffusion of error-resilient applications, research in Approximate Computing has flourished in the past few years, spanning through

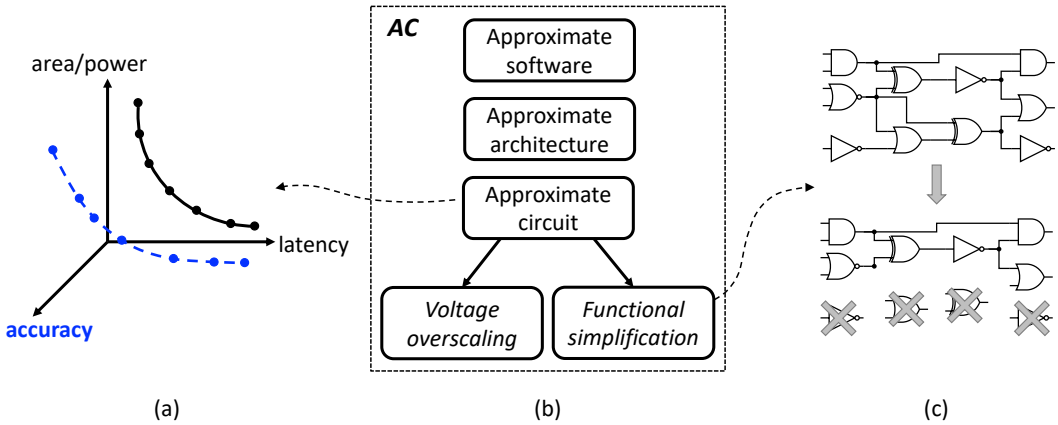


Figure 1.1. (a) Accuracy represents a new dimension for circuit synthesis . (b) Approximate Computing (AC) techniques taxonomy. (c) A simple example of functional approximation for circuits: some gates, along with their wired connections, are removed from the original circuit (above), resulting in a smaller inexact circuit (below).

different levels of the hardware/software stack [6, 7]. A first, rough division of AC techniques is represented by the three main categories of Figure 1.1b: approximate software development, approximate architecture exploration, and approximate circuit design [8].

An example of software, or program-level approximation [9,10] is loop perforation [11], where some iterations of a loop are skipped, either deterministically or randomly; the degree of perforation influences the accuracy of the final result. Another example belonging to the same category is thread fusion, where different threads are combined by merging two dynamic instances of the same static instruction into a single one [12]. Pattern reduction [13] is yet another possibility: here, different patterns which are usually found in parallel programs, such as *map* or *reduction*, are identified and associated with a specific approximation to optimise the overall performance.

At the architectural level, approximate storage and processors may be employed. Approximate processors architectures can either run traditional code on a processor designed to execute some specific instructions in approximate mode, or turn approximable segments of traditional code into a neurally inspired algorithm running on accelerators. In both cases, a compiler or manual intervention by a programmer are necessary to annotate approximable code segments [14, 15]. Memory and storage can also be exploited to trade accuracy for performance [16–18] by relaxing the requirements on storage precision, protec-

tion from particle strikes or RAM energy supply.

Finally, in approximate circuits, hardware is intrinsically designed to compute inexact operations. My research focuses on this level of approximation, which is further detailed in the next section.

1.1 Approximate circuits

Traditionally, circuit design has adopted several strategies to balance the trade-off existing between area (or power) and latency. Quicker circuits can hardly be obtained without increasing their size and their global power consumption. Approximate Computing adds a third dimension to the area-latency trade-off: the accuracy of the circuit output (Figure 1.1a), offering a substantial number of new possibilities to designers and engineers struggling with their demanding requirements.

The large family of approximate circuit design techniques can be divided into the two subcategories of Figure 1.1b: 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 [14, 19, 20]. However, these timing errors may result in uncontrollably large computational errors, limiting the usability of these solutions [8]. Workaround to this problem are redesign techniques, such as the ones proposed in [21], or careful considerations on the statistical distribution of inputs [22].

In functional approximation, instead, the original boolean function executed by a circuit is modified with the purpose of trading accuracy for performance. Figure 1.1c illustrates a simple example: some gates of a generic circuit are cut to obtain an approximate version, which will compute erroneous values for a subset of the circuit inputs. If this error is limited, and can be tolerated by the application of interest, the original circuit can be replaced by its smaller, more efficient approximate version [1, 23].

Many approaches belonging to this category focus on the design of specific approximate arithmetic units, e.g. adders [24–29], multipliers [30–35], or dividers [36–38]. Other works [39, 40] present algorithms that allow to automatically explore the energy-quality trade-off, but again limit the analysis to adders and multipliers only. Unlike these hand-crafted or circuit-specific designs, many research contributions developed wider-scope generic approximate circuit design techniques, that can be applied to any logic circuit without a-priori knowledge of its functionality [2–5, 23, 41–44]; my research work also belongs to this latter category.

Moreover, these circuits constitute the basic building blocks for large-scale applications [45–47] such as, for instance, neural networks [48–51], and several works study how the error can be distributed [52] or can propagate [53–57] among different approximate circuits in a larger system.

1.2 Approximate Computing potential: a case study for multiple approximation levels

In Agrawal *et al.* [58] the authors explore the combined effect of approximations introduced at different levels, in order to demonstrate the potential of Approximate Computing in enhancing the performance of error-resilient applications.

To represent different approximation levels, the authors chose to apply simultaneously loop perforation, reduced arithmetic precision and relaxation of thread synchronisation. In reduced precision, variables and data structures of a program are represented with fewer bits than the original integer or floating point representation, allowing to employ smaller circuits and, hence, using cheaper and less consuming hardware. Thread synchronisation instead is employed to guarantee that different threads reach execution checkpoints in a predictable manner, or share the correct values of global variables; however, synchronisation can be systematically relaxed to improve performance.

The three above mentioned approximation techniques are applied to common image processing applications, such as Synthetic Aperture Radar (SAR) and Wide Area Motion Imagery (WAMI), or commonly used machine learning techniques, such as K-means Clustering and Deep Neural Networks, and, finally, Robot Localization.

Their results confirm and validate the interest in Approximate Computing: indeed, without loss of accuracy (which means detecting the correct objects in image recognition, or identifying the same clusters in K-means), they were able to perforate hot loops with an average factor of 50%, with a proportional decrease of the execution time. Moreover, data bit width was reduced from 32 or even 64 bits to a range of 10 – 16 bits. As for parallel applications, they were able to halve the execution time through relaxed synchronisation. However, the most important result is that concurrent application of these three techniques further improved the overall performance.

Their conclusion, which I strongly agree with, is that "*As the benefits of approximate computing are not restricted to a small class of applications, these results motivate a re-thinking of the general purpose processor architecture to natively sup-*

port different kinds of approximation to better realise the potential of approximate computing.".

My PhD research work has focused on approximations at hardware level, always with a view to a subsequent integration in larger systems, where potentials of approximations are exploited at *all* levels. Indeed, works as [58] further encourage research in approximate hardware synthesis, since they prove that no level has more potential than the others, but rather the wise combination of approximations may represent a key to sustainable development.

Chapter 2

Approximate Logic Synthesis: algorithm categorization and error modeling

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

Approximate hardware components realized with ALS can, at the same time, offer remarkable gains in area and efficiency *and* significant performance increases with respect to their exact counterparts, in exchange for small losses in output quality. ALS is hence the embodiment, at the hardware design level, of Approximate Computing (AC). Approximate circuit synthesis is particularly attractive since approximate circuits are employed as basic blocks for realizing application-specific accelerators, which are a highly relevant component of modern Systems-on-Chips [59].

The large family of algorithms that functionally modify a circuit are reunited under the domain of Approximate Logic Synthesis (ALS).

2.1 ALS algorithms

The transformation of a generic Boolean function f into its approximate counterpart $\tilde{f}$ can be performed in different ways.

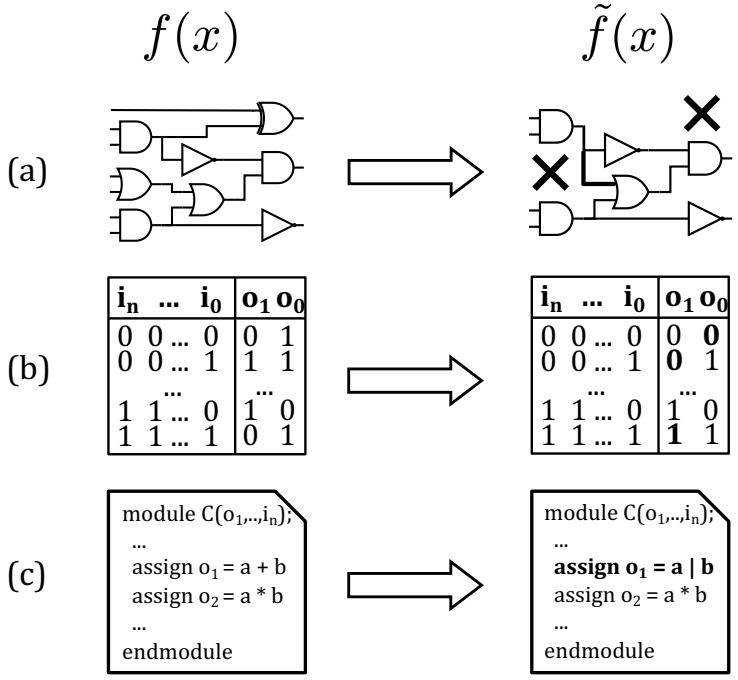


Figure 2.1. Possible functional simplification approaches are illustrated: a generic Boolean function f is transformed into $\tilde{f}$, either acting on a synthesised netlist, for instance at gate-level (a), or on its truth table (b). Finally, the circuit at behavioural level can be simplified by approximate high-level synthesis (c).

Three main classes can be identified for such approaches, illustrated in Figure 2.1: *Netlist transformation*, *Boolean rewriting* and *Approximate high-level synthesis*.

In **Netlist transformation** (Figure 2.1.a), the Boolean function f is already mapped into a netlist, *i.e.*, a list of components such as Boolean gates implementing simple functions, for instance AND, OR and NOT. Approaches belonging to this category start from these netlists, and transform them by removing some nodes, or by substituting some wires with others, hence reducing the circuit size and power consumption. My research work has focused on this category of techniques.

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 input combinations of f and the corresponding outputs is available. Therefore, this description is independent from the netlist

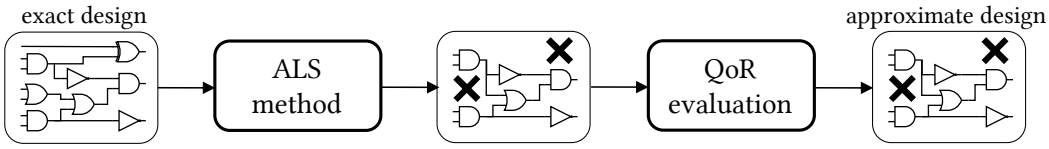


Figure 2.2. Common ALS flow in the state of the art: after the application of an ALS method, the approximate circuit is evaluated to verify that the induced error does not exceed the predefined limit. The result of this test is the final approximate design.

selected to implement the function, generating a specific circuit. Methods belonging to this category, illustrated in section 2.1.2, modify the values of such outputs for a subset of the input combinations, as illustrated in Figure 2.1.b: in the output columns, some values in bold have been flipped w.r.t. the original truth table.

Finally, **Approximate high-level synthesis** focuses on the highest level of abstraction for ALS, where the function is described at behavioural level, such as in RTL Verilog or C language. An example fragment of such code is depicted in Figure 2.1.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.

Independently from the approach chosen among those described above, the process of synthesizing an approximate circuit is exemplified in Figure 2.2: an ALS method is applied to the exact design, leading to a smaller – and more efficient – circuit. In the simple example of the figure, representing a netlist transformation approach, two gates are removed from the circuit. Before obtaining the final approximate design, a QoR evaluation phase is necessary: the circuit needs to be tested in order to verify that the error does not exceed the tolerated limit. We will see that an earlier error modeling phase is fundamental for predicting the effect of functional transformations and guide ALS methods: such preliminary step represents a major focus of my work.

The next sections report the principal works in the state of the art for the two first algorithm classes of Figure 2.1, namely netlist transformation and Boolean rewriting. Approximate high-level synthesis, instead, will not be further investigated, since its higher level of abstraction makes it less relevant in the discussion of this thesis.

2.1.1 ALS structural netlist transformation

Among methods that implement structural netlist transformation, three main categories can be distinguished: greedy heuristics, stochastic netlist transformations, and exhaustive exploration.

Shin *et al.* [60] employ a greedy strategy for generic circuit simplification, where a set of stuck-at-faults are injected in the circuit, and a 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.

GLP by Schlachter *et al.* [1] presents another greedy iterative algorithm for circuit simplification. The proposed framework, depicted in Figure 2.3, is simple but effective. The exact circuit is represented as a direct acyclic graph and nodes are pruned according to two main criteria: the node significance, which represents the impact of that node on the final output, and the node activity or toggle count. According to the application characteristics, nodes can be pruned starting from those with lower significance, lower activity, or a combination of the two: the significance-activity product (SAP). Node activity is obtained through gate-level hardware simulation, while significance is computed in a reverse topological graph traversal, starting from the primary outputs arithmetic bit-significance, then assigning to each node i the significance σ_i:

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

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

After nodes have been ranked according to the desired metric, the GLP framework iteratively removes a node from the original circuit, setting its output to a constant; it then resynthesizes the circuit and simulates it with a Monte Carlo process to verify that the error constraints on error rate and mean relative error have not been violated. This, indeed, represents the QoR validation phase depicted in Figure 2.2. After each iteration of gate removal, GLP recomputes SAP for node ranking, until the error threshold is reached.

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

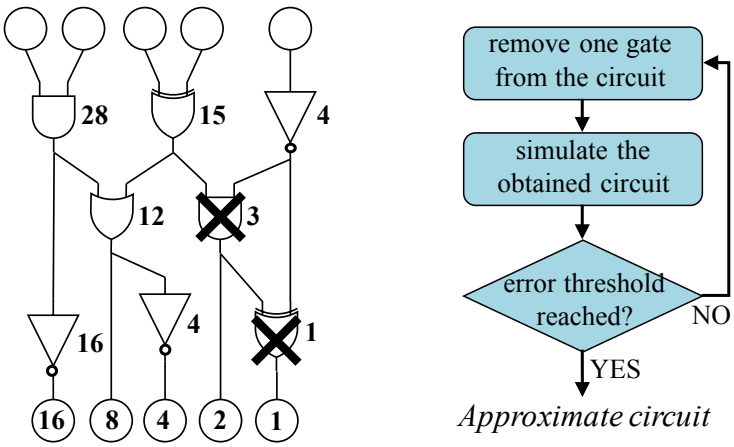


Figure 2.3. GLP [1] framework. Gates are removed from a generic circuit starting from the least significant, until the allowed error threshold is reached.

Venkataramani *et al.* [2] propose another greedy strategy called SASIMI (Substitute And SIMplIfy). In SASIMI, functional approximation is performed by identifying pairs of signals that assume the same value with high probability, and substitute one with the other. When the target signal is replaced, the gates belonging exclusively to its generating cone of logic are removed from the circuit, while others can be downsized, as illustrated in Figure 2.4. Clearly, the error induced by a potential substitution must be considered in the choice of the target signal. This error can be estimated through a Monte Carlo process that assess the error rate and average absolute error magnitude on a subset of all possible circuit inputs, which are assumed to be uniformly distributed. The algorithm takes as input the original circuit and a target error, then iteratively performs the selection of the best candidate signal pair, the substitution and consequent circuit simplification, followed by QoR evaluation. Once the target error constraint is reached, the iterative algorithm stops.

Liu *et al.* [61] argue that the assumption of uniform distribution of input data is seldom correct. Therefore, they propose SCALS: Statistically Certified Approximate Logic Synthesis, an iterative framework where statistical hypothesis testing is employed to estimate the errors obtained on the circuit outputs after an approximate transformation. This approach guarantees that the population behaviour is indeed a faithful representation of the actual data distribution. 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.

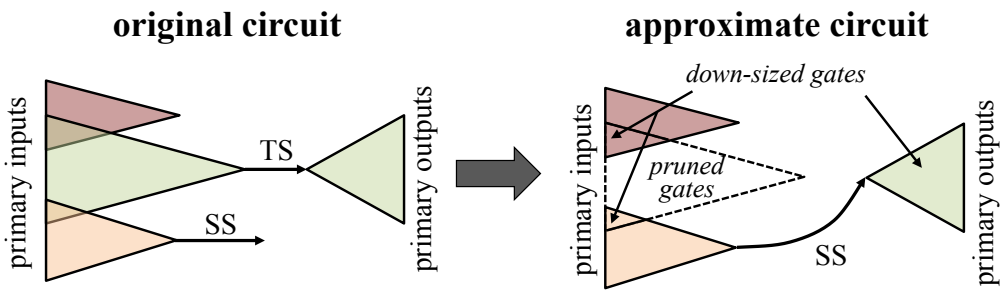


Figure 2.4. In SASIMI [2], a target signal (TS) is substituted with another circuit signal (SS). Gates belonging to TS cone of logic are either removed or downsized.

In a similar direction, Vasicek and Sekanina propose EvoApprox, a genetic algorithm to mutate the circuit into approximate versions by swapping gates with wire connections [62]. Circuits are represented as direct acyclic graphs, whose nodes can be Boolean gates or more complex components according to the technology library chosen. The nodes are contained in a two-dimensions grid, the *chromosome*, which is randomly modified to explore new design points. This mutation evolves using a fitness function, which leads to better approximation over runtime. After computing area and error of the initial population, the algorithm iteratively selects the best-scored circuit, generates λ offspring from the parent through mutation, and evaluates the new population. 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.

Using And-Inverter-Graphs as the circuit representation, Chandrasekharan *et al.* [63] propose an algorithm for approximate AIG re-writing that guarantees the bounds of approximation errors introduced in the approximate circuit. First, the critical paths are identified in the AIG, where the critical paths are the paths from the primary inputs to the primary outputs with the largest number of nodes. After this step, cut enumeration on the selected paths is used to identify potential cuts. A SAT solver is then employed to compare the original AIG and the approximate AIGs to check wether the error constraint is violated, hence guaranteeing the error bound.

As opposed to other works surveyed in this section, Circuit Carving [23] does not employ iterative approximations towards inexact logic synthesis. It instead resorts to exhaustive exploration of all possible nodes subsets that can be re-

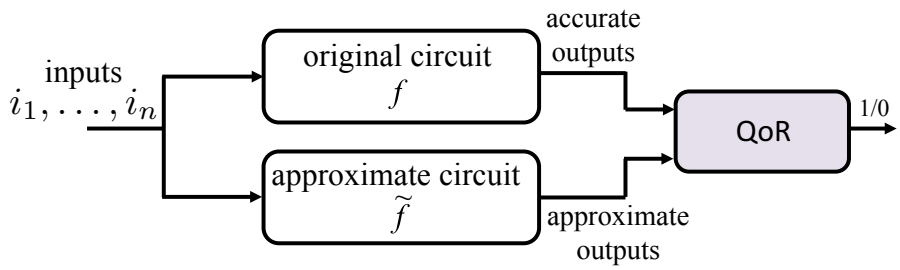


Figure 2.5. In SALSA a QoR circuit is constructed to compare the outputs of exact and approximate circuits [3]. Observability *don't cares* of the approximate circuit are used to minimize the approximate circuit logic.

moved from the exact circuit, among which the most convenient will be chosen. The best candidate sub-circuit is the largest one (in terms of number of gates) that does not overcome the identified error threshold. This latter work will be described in detail in Chapter 3.

2.1.2 ALS logic rewriting based methods

ALS using *Boolean rewriting* can be categorized as a general approach in which the logic of the circuit is first captured in a formal Boolean representation, then it is manipulated to yield an approximate Boolean representation; this, in turn, is synthesized to a gate-based netlist. In this approach, the approximations are captured into Boolean expressions too, then employed to relax the Boolean minimization of the original circuit [3, 64, 65].

One of the earliest works to employ this approach is SALSA [3]. In SALSA, a *QoR circuit* is first constructed by comparing the outputs of the original circuit and the approximate circuit using a comparator, as depicted in Figure 2.5. 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 – in short, approximation effects do not impact the final QoR. This set of *don't cares* can be then used to minimize the approximate circuit using standard logic synthesis techniques [66].

SALSA has been extended in ASLAN [67] to handle sequential circuits, where errors arise over multiple sequential cycles. ASLAN employs a circuit block explo-

ration method to identify the impact of approximating the combinational blocks, 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 [65] propose a local minimization approach to simplify a circuit by substituting the Boolean expressions of the internal circuit nodes with approximate expressions that require less logic (for instance, by dropping some literals). Since a circuit can have millions of internal expressions, each with a number of possible ways to re-write approximately, a knapsack formulation is constructed and solved to identify the best set of nodes to approximate in order to maximize value (*i.e.*, total area reduction) under weight constraints (*i.e.*, maximum error).

BLASYS [4] introduces a new formal method in approximate logic synthesis [68, 69]. In BLASYS the operation of a circuit is captured by a matrix that represents the output side of the circuit's truth table. To create an approximate circuit from a given circuit, Boolean matrix factorization is used, where an input matrix M is factored into two matrices B and C such that $M \approx BC$ and the distance $|M - BC|_2$ is minimized. In Boolean Matrix Factorization (BMF), multiplications are performed using logical AND operations and additions are performed using logical OR operations. 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.

Since the construction of the matrix that represents the truth table is limited by the number of primary inputs of the circuit, BLASYS incorporates a hypergraph partitioning method that breaks down a large circuit into a number of subcircuits, so that BMF can be applied to each subcircuit. The original subcircuit is substituted by its approximate version, then, the QoR and design area or power are evaluated for the entire circuit, as displayed in Figure 2.6. Monte Carlo evaluation is used to identify which subcircuit is most resilient and, hence, should be factorized first.

Table 2.1 summarises and compares the ALS techniques described so far.

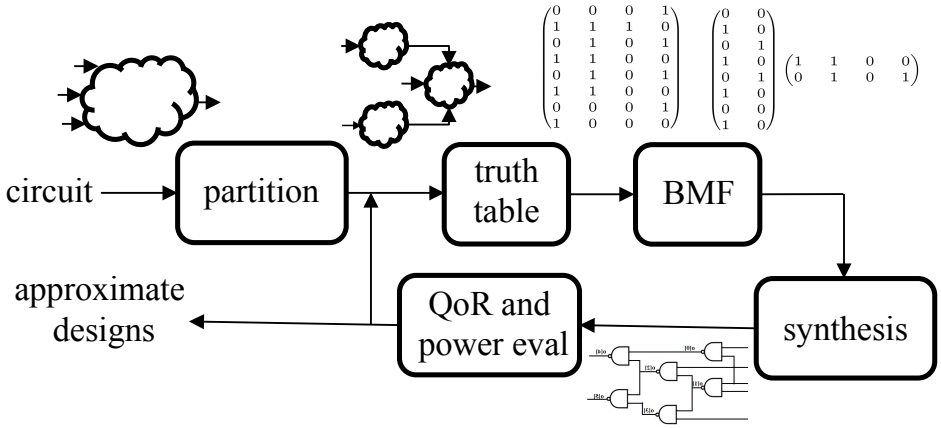


Figure 2.6. BLASYS flow [4]. An input circuit is first partitioned into subcircuits with a reasonable number of inputs for each subcircuit. The truth table of every subcircuit is then evaluated followed by Boolean Matrix Factorization (BMF), the result is then synthesized to create an approximate subcircuit. The QoR and physical metrics (e.g. power or area) of the entire circuit are then evaluated.

2.2 Error modeling for guiding ALS

In the past section, the most important and commonly used ALS strategies were described. My PhD work also focused on an additional aspect of approximate circuits design: that of *error modeling* for guiding such strategies.

Figure 2.7 illustrates the Approximate Logic Synthesis flow that is followed in this thesis, which augments and improves the state of the art depicted in Figure 2.2. Here, the ALS core method is preceded by an *error modeling* phase.

Indeed, before applying a given approximation to an exact design, it is essential to estimate how much that transformation will impact on the final result. Therefore, an error modeling phase must be present, with the aim of annotating a circuit (or a Boolean) specification with a notion of error. 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 *guides* ALS methods in identifying the least error-prone transformation, or set of transformations.

The importance of the presence of the error modeling phase is showcased in Figure 2.8, which illustrates the performance of an approximate circuit obtained by the GLP method [1] when guided by an accurate error modeling algorithm,

Table 2.1. Summary of the ALS works described, where NT stands for Netlist Transformation, and BR for Boolean Rewriting.

Reference	**Cat.**	**Description**
GLP [1]	NT	Iterative gate removal, employs (conservative) significance to rank nodes
SASIMI [2]	NT	Iterative substitution of signals with similar behaviour; Monte Carlo employed for error estimation
SCALS [61]	NT	Stochastic exploration with hypothesis testing for error estimation
EvoApprox [62]	NT	Iteratively introduces random mutations into the circuit
Chandrasekharan *et al.* [63]	NT	Employs approximate AIG re-writing by identifying critical paths that can be shortened
Circuit Carving [23]	NT	Exhaustive exploration for search of optimal solution, coupled with exact pruning criteria
SALSA [3]	BR	Exploits observability *don't cares* to simplify the original circuit
Wu and Qian [65]	BR	Substitutes logic expression with approximate ones that require less logic
BLASYS [4]	BR	Exploits matrix factorization and subsequent simplification

called PP-Mon [5], as opposed to its original implementation, which instead employs a highly conservative error modeling strategy. Indeed, when guided by PP-Mon, GLP retrieves much smaller, faster and more efficient circuits for the same amount of tolerated error, as expressed by the Energy, Delay and Area Product (EDAP) on the y-axis.

Several ALS algorithms do not rely on the error modeling phase, such as Vasicek *et al.* [62], where a genetic programming technique is employed to randomly mutate the circuit and explore the search space without *a priori* notion of the induced error; however, the execution time and final QoR of these algorithms could highly benefit from an accurate error modeling phase.

Other works instead, such as Circuit Carving [23], compute a tight bound on maximum error in the first phase, and then the resulting simplified circuits do not need to undergo the QoR evaluation phase, as they are already guaranteed to not overtake such bound.

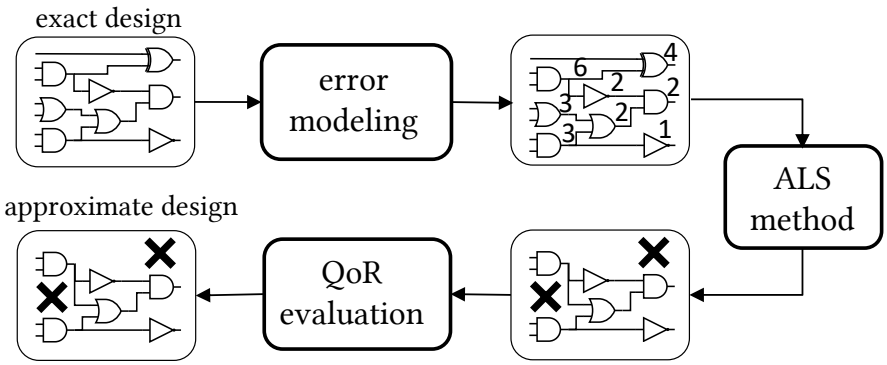


Figure 2.7. A phase of error modeling can precede Approximate Logic Synthesis, with the aim of decorating a circuit (or a Boolean specification) with a notion of error, and hence guiding subsequent logic-simplification decisions. Then, the phase of QoR evaluation completes the process to verify whether output quality constraints are satisfied in the synthesised approximate circuit.

In the next section, the most commonly used error metrics for both error evaluation phases will be described, since their formal definition allows to understand precisely how different algorithms of the state of the art approach error estimation. Next, the principal strategies for error modeling of the state of the art are described.

2.2.1 Error Metrics

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

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

Then, an input-independent distance D must be derived from all values of d according to a metric. Alternative choices for such metric are influenced by several factors: the nature of the application in which the approximate hardware will be employed, its criticality, etc. For example, Ma *et al.* [69] and Venkataramani *et al.* [3] employ the Hamming Distance as a measure of d, defined as the number of bit flips in $\tilde{f}$ w.r.t. the original f.

The most common, widespread metrics for D employed in the field are now

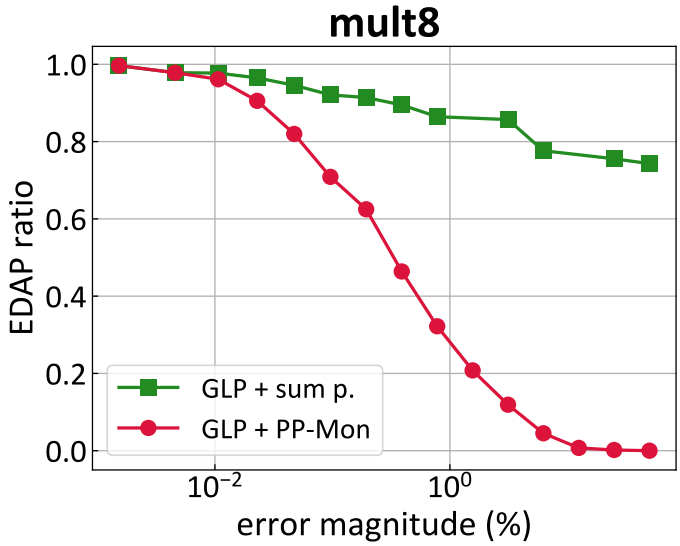


Figure 2.8. Energy, delay and area (EDAP) reduction for different error constraints in an 8-bit multiplier. For the same level of tolerated error, GLP guided by PP-Mon (red) obtains much smaller, faster and more efficient circuits than its original implementation (green), which is guided by a simpler error modeling approach, called sum propagation.

defined. Referring again to the Hamming distance, one could be interested in the maximum or average Hamming distance over the inputs of f.

When, as done in [1,5,23,63], the focus is on controlling the Maximum Error (i.e., worst case distance) that occurs when a circuit is approximated, D is defined as:

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

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

Several Approximate Logic Synthesis techniques [1,2,4,61,65,70,71] monitor average case distance (Mean Absolute Error) induced on the output, instead of focusing on potential outliers, expressed as

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

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

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

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

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

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

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

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

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

It is of course possible for a given Approximate Logic Synthesis approach to consider more than one error metric. Indeed, in [3, 42, 60, 63, 64, 67, 72], both maximum error and average error are taken into account.

Some works introduce further metrics for error estimation that also consider the structure of the circuit being simplified, in addition to its functionality. Notably, Zhang *et al.* [73] adopt the notion of Approximate Efficiency, defined as the ratio between the gain (in terms of Energy-Delay Product, EDP) deriving from the simplification of a node and the corresponding induced error: $\Delta EDP/D$. The underlying assumption is that, if two nodes generate the same error in the final output when pruned from the original circuit, the one leading to higher benefits should be pruned first.

The works described in Chapter 3 and 4 mainly focus on maximum error estimation and control. However, Chapter 5 describes how these methods can be adapted to evaluate mean absolute error.

2.2.2 Methods for Error Modeling

Finally, there exist several possibilities to quantify the error metrics introduced above, each presenting some strengths and weaknesses:

Exact error evaluation. A precise computation of the error caused by a circuit modification requires exhaustive evaluation for all possible input combinations. This number grows exponentially with the number of inputs, and hence such computation cannot scale to large circuits. SAT-solver based techniques were proposed [42] in order to help accelerating exhaustive evaluations; however, exhaustiveness necessarily becomes intractable at some point, as the circuit size increases. Hence, the need arises for methods to efficiently calculate error estimates (in the case of average errors and error rates) and error bounds (for maximum errors).

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 [42], resulting in an unbiased statistical estimate of D. Nonetheless, even a Monte Carlo implementation can become computationally intractable for large circuits if carried out in a straightforward way, because it necessitates distinct evaluations, at each simplification step, for all candidate approximate transformations. Su *et al.* [74] devise a technique that effectively lowers the computational effort entailed. The complexity of their algorithm, based on change propagation matrices that track the effects of an internal node to the primary outputs, is $\mathcal{O}(MOT)$, where M is the size of the Monte Carlo input set, O the number of primary output, and T is the size of the set of possible approximate transformations. A naïve Monte Carlo alternative has complexity $\mathcal{O}(MNT)$, where N is the number of nodes in the circuit, which is usually much larger than the number of its primary outputs. However, T grows exponentially with the size of the circuit and, hence, for large circuits it may still be too heavy to derive accurate estimates of the error.

Bounding maximum errors. Monte Carlo-based approaches are not employable when maximum error thresholds must be provided, as they can not account for outliers. As described in Section 2.1.1, GLP by Schlachter *et al.* [1] introduces an algorithm to compute error guarantees, which assigns to each node in a circuit the sum of the significance of all its reachable outputs (where the significance of the output bit i is equal to 2^i). This strategy is overly conservative, as it assumes that a node simplification can affect all reachable outputs simultaneously for at least one input combination (all ones becoming zeros, and vice-versa). In fact,

masking effects very often reduce the magnitude of perturbations caused by inexact transformations, preventing all outputs to assume erroneous values at the same time.

Chapter 4 summarises the main characteristics of these methods, along with the presentation of an approach I devised that improves on the state of the art by concentrating the strengths of the three, and limiting their weaknesses.

The next chapter, instead, illustrates my work Circuit Carving, an ALS method that employs exact evaluation to guide structural netlist modification. The accuracy on the induced error estimation combined with the exhaustive nature of the algorithm generate highly performant approximate circuits.

Chapter 3

Circuit Carving: exhaustively exploring approximations

3.1 Introduction

One of the first works of my PhD is an ALS algorithm called Circuit Carving (CC). This algorithm belongs to the category of structural netlist transformation, described in Chapter 2, but it presents a fundamental difference w.r.t. the other techniques in the state of the art: instead of employing heuristics for netlist modification, it resorts to pruned exhaustive exploration in order to identify optimal solutions.

Having as input a generic combinatorial netlist, Circuit Carving identifies its largest sub-circuit that can be discarded, *carved out* of the original one, without violating a user-specified error constraint. As opposed to several works, which are based on iterative simplifications on the input circuit, the approximation problem is casted as a binary search tree exploration which exploits accurate knowledge of the circuit gate-level error distribution. While this formulation has exponential complexity, the scalability of the proposed method is improved by implementing search-pruning criteria that exploit the circuit topology in order to effectively reduce the exploration time.

The key contributions of this work can be summarised as follows:

- The description of a novel methodology, called Circuit Carving, for the design of inexact digital circuits, which explores the search space of the exact circuit subsets.
- The introduction of pruning conditions that can effectively decrease the mentioned search space, and hence increase the framework scalability, by

disregarding search space regions that cannot contain candidate to optimal solutions.

- The employment of induction on identical blocks for error modeling of large circuits.

3.2 Problem formulation

The digital logic implementing a boolean function can be represented as a Direct Acyclic Graph (DAG) (N, E), where each node $n_i \in N$ represents a gate. An edge $(n_i, n_j) \in E$ represents a connection from node n_i to node n_j, in that the output of node n_i is used by node n_j.

Nodes are annotated with weights w_{n_i}, indicating the maximum difference visible at the circuit outputs, when setting the output of the node n_i to a constant value. The weight of an edge is equal to that of its destination node.

A cut (N_C, E_C), with $N_C \subset N$, $E_C \subset E$, identifies a subgraph of the original graph. An example of cut is shown by the dashed line in Figure 3.1a. The set $O(C)$ of the outgoing edges of a cut is defined as

$$O(C) := \{(n_i, n_j) \in E_C \mid n_i \in N_C \wedge n_j \notin N_C\}$$

If the outgoing edges of a cut are set to a constant value, 0 or 1, then the circuitry contained in the cut does not contribute to the circuit logic anymore, and the cut can therefore be suppressed. The resulting graph is, hence, a candidate inexact circuit.

The weight of a cut w_C is defined as the sum of the weights of its outgoing edges, and represents the maximal error introduced by removing from the exact circuit the gates corresponding to the nodes in C. In Figure 3.1a, cut C_1 has two outgoing edges, with weight 8 and 4 respectively, therefore $w_{C_1} = 12$; the methodologies that can be employed to derive such weights are explained later on, in Section 3.2.2; for now one shall assume these values known.

Circuit Carving (CC) aims at finding the largest cut (N_C, E_C) such that $w_C \leq T$, where T is a pre-defined error threshold. In other words, find the cut with the maximum number of gates whose removal from the exact circuit does not incur in a maximum error exceeding T.

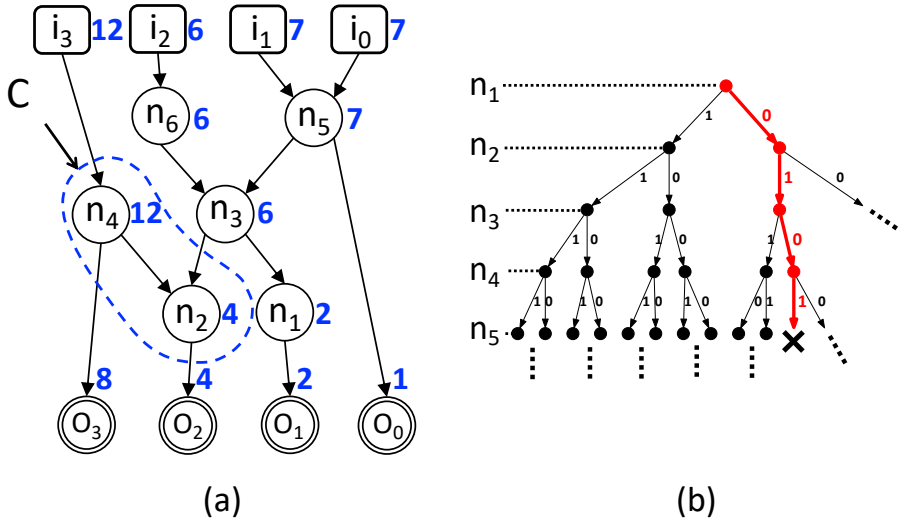


Figure 3.1. a) Example of a graph, and of a cut C within it. Labels besides nodes represent their weight. The weight of cut C is 12 (sum of the differences of its outgoing edges). b) The binary search path corresponding to cut C, outlined in bold, red. If $w_C > T$, the search in this path stops and the algorithm backtracks to the previous node.

3.2.1 Exploration Algorithm

To find such cut, the algorithm explores a search tree representing all subsets of graph (N, E). Each level of the exploration represents the inclusion (1-branch) or exclusion (0-branch) of node n_i in a cut. Therefore, each cut (N_C, E_C) is represented by its corresponding path in the binary tree; Figure 3.1b shows the path corresponding to cut C of Figure 3.1a.

Since this formulation has worst-case exponential complexity, pruning criteria are necessary to reduce the search space and obtain a more scalable framework.

A cut is *valid* if its weight w_C is not larger than T. Validity can be exploited as a pruning criterion when node exploration is performed in reverse topological order, i.e., for all edges $(u, v) \in E$, v is explored before u (as done in Figure 3.1a). In this way, w_C monotonically increases w.r.t. the exploration depth, because any outgoing edge of a cut cannot be recovered. If at some level i the cut weight overcomes the pre-defined error threshold T, the inclusion of further nodes in the cut cannot result in a valid candidate, and the search can be stopped (as in Figure 3.1b, considering an error threshold T smaller than 12). This criterion is called *validity* and its effectiveness is quantified in Section 3.3.2.

The second pruning criterion is derived from a novel property, which we call

closure, of a cut. Let us define the set of all parents of node n: $P_n = \{x \mid \exists(x,n) \in E\}$, and the set of all children $Q_n = \{x \mid \exists(n,x) \in E\}$. A cut is closed if for all nodes of the original graph it holds that:

$$\begin{cases} \forall x \in P_n \,.\, x \in N_C \Rightarrow n \in N_C \\ \forall x \in Q_n \,.\, x \in N_C \Rightarrow n \in N_C \end{cases}$$

In other words, in a closed cut, if all children of a node n belong to the cut, n must also be in the cut. Likewise, if all parents of a node n are in the cut, n is in the cut as well. Figure 3.2a depicts an example of a cut (C_2) that is *not* closed.

A cut that is not closed *cannot* be the solution to the Circuit Carving problem, because there always exists a larger cut with the same weight. Such (larger) cut is, indeed, called *closure*: formally, the closure $CL(C)$ of a cut is the minimal closed subgraph of G containing the cut: $N_C \subseteq N_{CL(C)}$, $E_C \subseteq E_{CL(C)}$. Figure 3.2a depicts the closure of cut C_2, $CL(C_2)$.

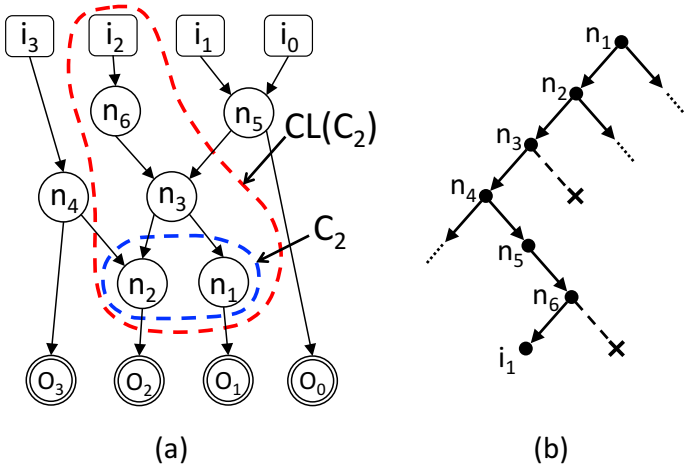


Figure 3.2. a) Cut C_2 is not closed, and cut $CL(C_2)$ represents its closure. b) Exploration path for cut C_2. Since $n_3 \in CL(C_2)$, branch 0 from n_3 in Figure 3.2b is not explored.

The concept of closure is crucial for reducing the exploration space: only closed cuts must be considered as candidates, since non-closed cuts are, by definition, sub-optimal. Therefore, the computation of closure $CL(C)$ can be exploited to prune the search. In fact, whenever the algorithm is exploring a non-closed cut C, at level i, it computes its closure and verifies whether the next node n_{i+1} in the search belongs to $CL(C)$. If it is so, it means that the current cut *must* be expanded to include node n_{i+1}, otherwise it could never represent a solution

to the problem. Hence, the 0-branch at node n_{i+1} can be disregarded. An example of this criterion at work is shown in Figure 3.2b, where the 0-branches at nodes n_3 and n_6 are not explored. This pruning criterion is called *closure*, and its effectiveness is also quantified in Section 3.3.2.

The third and last pruning criterion considers the number of gates included in the maximum cut found so far. If, at some level i, 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. This is the *residual gain* pruning criterion and its effectiveness is also quantified in Section 3.3.2.

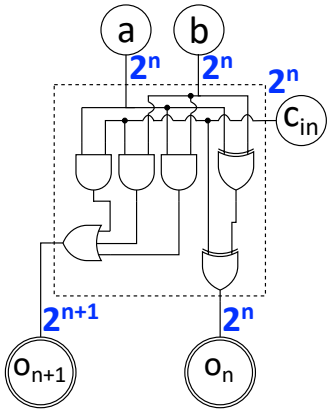
The three aforementioned pruning criteria contribute to reducing the binary search tree exploration run-time by orders of magnitude, making the CC framework more scalable. However, for large circuits and large error values, the complexity of the exponential search may still result in unreasonable execution times. In these cases, an upper bound can be added on the number of recursive calls to limit the execution time and, of course, foregoing optimality. Results in Section 3.3.1 demonstrate that, even when the number of recursions is bounded, the CC framework is tangibly more performant when compared to the state of the art.

3.2.2 Error modeling

A preliminary step to the binary search tree exploration consists in identifying and assigning weights w_n to all nodes $n \in N$. This process will be formally defined and described in Chapter 4 , which presents different strategies for error modeling. In this Section, I present one of these strategies, whose strengths and weaknesses will be discussed in Chapter 4.

First, output nodes weights are set to their bit significance (as can be seen in Figure 3.1a: output o_0 has weight 1, o_1 has weight 2, o_2 has weight 4 and so on) and then weights are propagated upward. A conservative way to implement upward propagation is, for every gate having fanout more than one (such as n_3 in Figure 3.1a), to compute the weight as the sum of the weights of its outgoing edges. This represents an upper bound for gate weight, since it would be impossible, when removing a gate, to produce a difference on the output larger than the sum of the bits reachable from that gate.

The *exact* values of weights for single gates, instead, can be obtained through exhaustive simulation. To do so, each gate is set to a constant value and the circuit deprived of that gate is simulated for all its input values; its output is then compared with the exact output, and w_{n_i} is set as the maximum obtained



(a)

abc_{in}	$2^{n+1}2^n$
000	00
001	01
010	01
011	10
100	01
101	10
110	10
111	11

(b)

Figure 3.3. a) A full-adder labelled with its weights. b) Its truth table.

error. While node weight values are exact, our definition of *cut* weight w_C as the sum of its outgoing edges is still a conservative estimate. This is because, as in the example of Figure 3.3a, in many practical cases the removal of a cut cannot influence all its reachable outputs simultaneously.

Whenever the number of circuit inputs is too large to allow exhaustive gate-level simulation, two possible strategies can be adopted. The first one consists in running the simulations on a random subset of the circuit inputs instead of the whole set, but this would invalidate the exactness of the error modeling method. Otherwise, the graph topology can be exploited to infer, when possible, weight-propagation rules.

An example of the latter strategy application is the full-adder, depicted in Figure 3.3a. Consider the last full-adder of a ripple carry chain (output weights, therefore, 2^{n+1} and 2^n); it can be observed from its truth table in Figure 3.3b that by setting to a constant value one bit in $\{a, b, c_{in}\}$, while leaving the other two unchanged, the maximum obtainable difference is 2^n. Therefore, the three inputs of the full adder are labelled with the weight of the less significant output bit, i.e., 2^n. Since this property holds for any of the previous full-adders of the carry chain, this weight can be propagated to derive the weights for a ripple carry adder of arbitrary length (Figure 3.4).

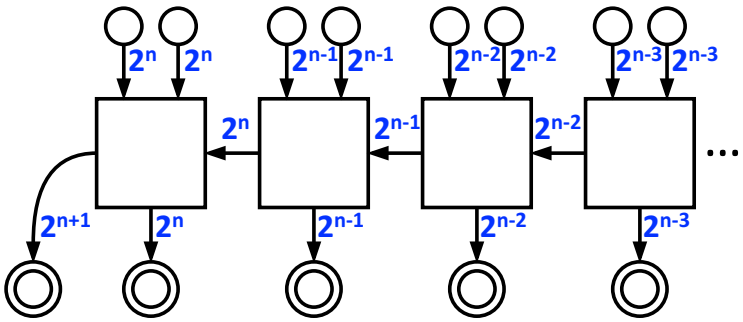


Figure 3.4. A 4-bit ripple carry adder with error modeling derived from weight-propagation.

3.3 Experimental evaluation

3.3.1 Performance of inexact circuits

To assess the performance of Circuit Carving, the approximate circuits obtained by it have been compared with those obtained by GLP [1], a greedy strategy belonging to the state of the art, and described in Chapter 2. In GLP, weights (called node *significance* in their work) are assigned through (conservative) sum propagation. Then, a greedy selection of nodes to be removed from the circuit is performed, and resulting netlists are iteratively simulated — for a subset of combinations of the inputs — to check for correctness. Nodes to be removed are chosen starting from those with lower weight. In practice, the GLP method corresponds to choosing *a single path* in the search space, without backtracking, as opposed to the entire exploration aware of the exact weights performed by Circuit Carving. This makes CC more computationally expensive, but also more effective, in terms of finding larger sub-circuits to be eliminated from the exact circuit.

The trade-offs between performance and maximum tolerable error, for the inexact implementations designed with CC and GLP, can be observed in Figure 3.5. The figure of merit on the vertical axis is the Energy-Delay-Area Product (EDAP), normalized with respect to the corresponding exact circuit. The choice of evaluating EDAP, as opposed to only area savings, is to provide a more comprehensive view on the result quality, as well as a fair comparison with the state of the art. The horizontal axis, *error magnitude*, indicates the maximum absolute error constraint, as a percentage of the maximum output of the circuit.

The figure showcases how the CC framework consistently achieves better results across all benchmarks when compared to GLP, i.e.: it derives circuits with smaller EDAP for the same error constraint. As an example, CC reduces the EDAP

of an 8-bit multiplier by 40% when an error magnitude of 1% (i.e., an error of 1% of the maximum output) is tolerated, while GLP only leads to a ∼10% EDAP reduction in that case.

Note that, for the smaller benchmarks processed in Figure 3.5, namely the 8-bit adders and the 4-bit multiplier, binary search tree exploration algorithm scaled for all possible error values. However, for the larger benchmarks, while exploration did terminate for small error values, it did not terminate — within a threshold of one hour of computation — for larger error values. The divide is shown by the vertical, dashed line in the graphs of the 32-bit adders and the 8-bit multiplier. For the points to the right of the vertical dashed line, where complete exploration was not performed, the best result achieved at the time of termination is reported. It can be seen that the CC methodology, even when stopped short of finishing the exploration, still finds larger subgraphs to be removed, resulting in approximate circuits with lower EDAP, when compared to GLP.

3.3.2 Effectiveness of Search Space Pruning

To quantify the effectiveness of the three search space pruning criteria (*validity*, *closure* and *residual gain*), the exploration algorithm was run on the 8-bit ripple carry adder benchmark, each time turning off some of the criteria, and the number of calls required to complete exploration in each case was plotted. Note that, since the criteria are exact, when they are turned off the solution found at the end of exploration is the same — but at the cost of more recursive calls being performed. Results are shown in Figure 3.6, which reports the number of recursive calls (capped at 10^9) required when different combinations of the three criteria were used.

The data shows that exploration does not terminate within 10^9 calls when the *validity* condition is not considered. Compared to *validity* alone and for an error magnitude of 5%, the combination of *validity* and *closure* results in a reduction of the search space of ∼10X, while applying *validity* and *residual gain* reduces it by ∼30X. When all three pruning criteria are used, a search space reduction of ∼600X is achieved. For larger errors, the difference between curves increases even more, to several orders of magnitude, and finally, for extremely large error values (which correspond to removing almost the whole circuit) exploration correctly converges extremely fast when the *residual gain* criterion is turned on.

This work was published as a full paper at the DATE (Design, Automation and Test in Europe) conference in 2018 [23].

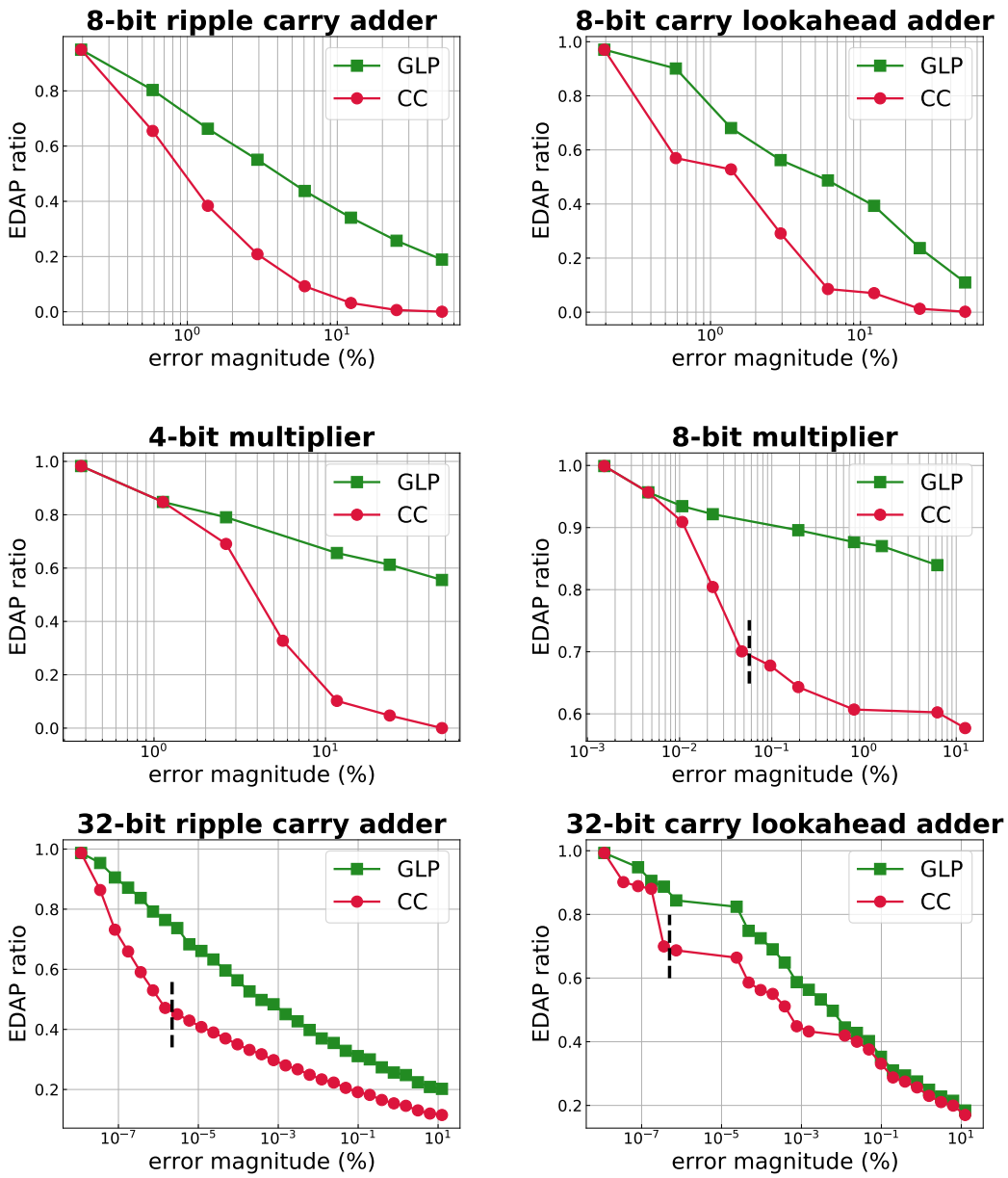


Figure 3.5. Energy-Delay-Area Product (EDAP) of the inexact circuits derived from our methodology and from the greedy approach of Schlachter et al. [1], varying the error constraint. EDAP values are normalized with respect to the exact implementations.

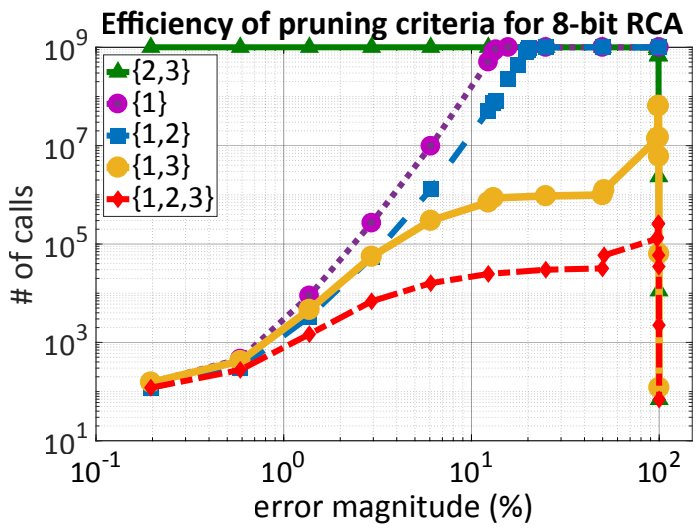


Figure 3.6. Number of recursive calls in the binary search tree exploration algorithm, when different combinations of pruning criteria are applied: 1) validity, 2) closure and 3) residual gain.

Chapter 4

A formal framework for error modeling

The previous chapters have introduced the idea that an accurate error model is fundamental for the performance of Approximate Logic Synthesis. Indeed, my latter research has focused on the definition and identification of algorithms for error modeling.

The strategies adopted so far to obtain an error model are of three different types: Monte Carlo sampling of the circuit inputs and simulation over the resulting input subset is a first possibility [2, 4, 74]. 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, as in Circuit Carving [23], or implicitly by employing SAT-solvers [42]. 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.

Finally, conservative bounds can be adopted to estimate such weights by using sum propagation, where node weights are assigned to the sum of all their children weights. However, such overly conservative weights have proven to provide poor guidance to the ALS method applied subsequently, as illustrated by my work [5], detailed later in this chapter. To sum up, each of these methods presents both strengths and weaknesses, outlined in Figure 4.1.

The approach proposed in this chapter introduces a powerful alternative to these methods. As in other state of the art algorithms, the circuit is represented as a Directed Acyclic Graph (DAG), and the final purpose is again to derive an error model which identifies bounds on the maximum error induced on the circuit output if a gate is removed from the circuit.

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

Figure 4.1. Summary of the main strengths and weaknesses of the state of the art techniques for error modeling.

However, the crucial step introduced 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, starting from the lower ones in a bottom-up traverse.

Figure 4.2 illustrates the described approach: 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 bottom-up, 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 propagation function. The envisioned framework is, hence, called Partition and Propagate (P&P).

4.1 Error model formal definition

Let us now repeat our model definition and expand it for the aim of illustrating the P&P framework: 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 $\boldsymbol{o} \in \mathbb{B}^k$,

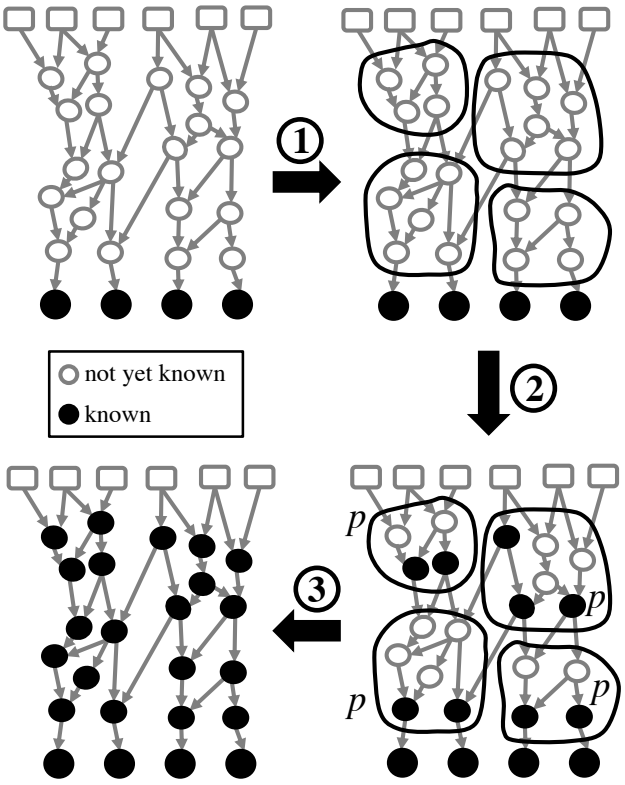


Figure 4.2. Different phases of the proposed error modeling approach. Initially, only output bounds are assigned, then the graph is partitoned (1), the propagation function is applied for all subgraphs, so that subgraph output nodes are labelled (2) and, finally, full simulation is performed in every subraph to label internal nodes (3).

so that $|I| = 2^h$ and $|O| = 2^k$.

The result of a circuit computation is captured by function

$$f : \mathbb{B}^k \to \mathbb{D} \tag{4.1}$$

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, one 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. It can be assumed that the node output is set to 0 without loss of generality, since

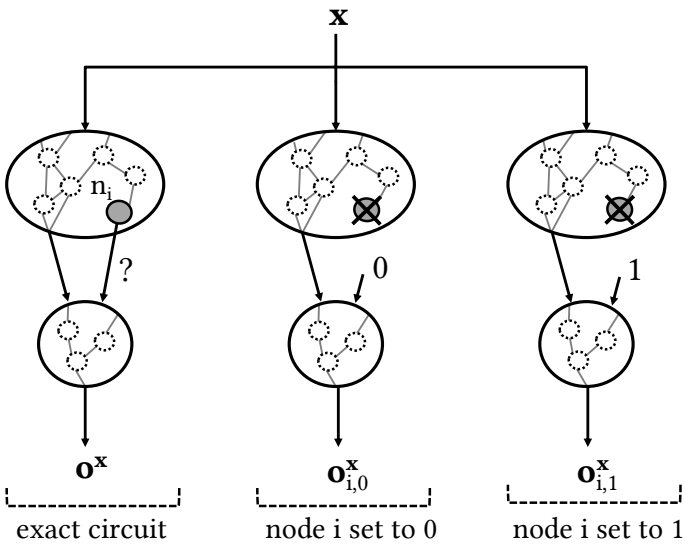


Figure 4.3. comparison between exact circuit and approximate versions with a gate set to 0 or 1.

the equivalent process can also be performed with the output set to 1.

Figure 4.3 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 $\boldsymbol{x}$, the quantity of interest is error

$$e_{\boldsymbol{x}} = f(\boldsymbol{o}^{\boldsymbol{x}}) - f(\boldsymbol{o}^{\boldsymbol{x}}_{i,0}) \tag{4.2}$$

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

$$e_{01,\boldsymbol{x}} = f(\boldsymbol{o}^{\boldsymbol{x}}_{i,1}) - f(\boldsymbol{o}^{\boldsymbol{x}}_{i,0}) \tag{4.3}$$

since

$$|e_{\boldsymbol{x}}| \leq |e_{01,\boldsymbol{x}}| \tag{4.4}$$

Equation (4.4) would hold even if the output of node n_i was set to 1: indeed, there is no difference in the choice of the constant replacing the node output. In any case, we are interested in estimating the maximum possible discrepancy from the exact primary output, which is represented by the absolute value of error $e_{\boldsymbol{x}}$:

$$\max_{\boldsymbol{x}} |e_{\boldsymbol{x}}| \leq \max_{\boldsymbol{x}} |e_{01,\boldsymbol{x}}| = \max_{\boldsymbol{x}} |f(\boldsymbol{o}^{\boldsymbol{x}}_{i,1}) - f(\boldsymbol{o}^{\boldsymbol{x}}_{i,0})| \tag{4.5}$$

where the maximum is taken over all possible inputs $\boldsymbol{x} \in I$. Error modeling is, then, the process of estimating (4.5).

A *partition* function is defined as

$$P : (N, E) \mapsto S \tag{4.6}$$

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 $(N_s, E_s) \in S$, therefore $\bigcup_s N_s = N$ and $\bigcap_s N_s = \emptyset$. Similarly, $E_s = \{(n_i, n_j) \in E \mid n_i \in N_s \wedge n_j \in N_s\}$.

External edges $\{(n_i, n_j) \mid n_i \in N_s \wedge n_j \in N_t \wedge s \neq t\}$ are 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 $n_j \in N_q$.

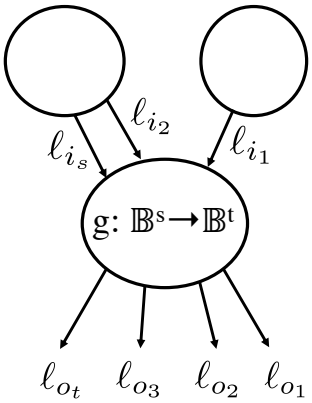


Figure 4.4. Label propagation model for a generic subgraph with s inputs and t outputs, implementing function g.

Figure 4.4 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 from function g:

$$p : L^t \times (\mathbb{B}^s \rightarrow \mathbb{B}^t) \rightarrow L^s \tag{4.7}$$

The process of deriving labels through the propagation function is performed in a bottom-up traverse of the graph, starting from assigning the primary output labels (for example, to their bit significance). Labels carry information on errors associated to approximate transformations, and the definition of set L depends on the chosen error representation. For example, the integer weights introduced in Chapter 3 for Circuit Carving are particular instances of labels. Section 4.3 showcases 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 4.5 illustrates that the framework presented in this thesis captures a whole family of existing error-modelling approaches, each with different characteristics. In particular, three techniques fit into the described framework: state-of-the-art sum propagation [1], and two Partition and Propagate versions developed during my PhD: P&P-Monotonicity (PP-Mon) and P&P-Intervals (PP-Int), illustrated in the next sections.

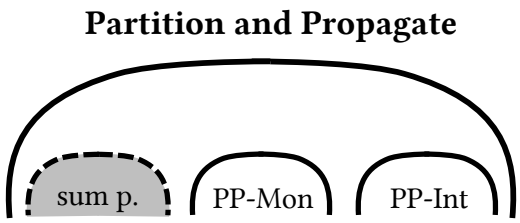


Figure 4.5. This formal framework subsumes multiple error-modeling approaches. Sum propagation belongs to the state of the art, while all the rest is a novel contribution.

4.2 P&P-Monotonicity

P&P-Monotonicity (PP-Mon) is an algorithm that combines partitioning into subgraphs and studies the functionality of each subgraphs: in fact, PP-Mon exploits subgraphs monotonicity to obtain accurate weights.

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. Although similar to the *unate* property of functions, monotonicity is preferred because it allows to distinguish between strict monotonic and non-strict monotonic, the importance of which will be clarified in the next sections.

4.2.1 Propagation mechanism

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

$$g : \mathbb{B}^s \rightarrow \mathbb{B}^t \tag{4.8}$$

where s is the number of inputs of the truth table, and t the number of outputs. Then, function

$$g_{n,v} : \mathbb{B}^{s-1} \rightarrow \mathbb{B}^t \tag{4.9}$$

is defined as

$$g^{\boldsymbol{x}}_{n,v} = g(x_1, ..., x_{n-1}, v, x_n, ..., x_{s-1}). \tag{4.10}$$

Equation (4.10) 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. $\boldsymbol{y} \in \mathbb{B}^{s-1}$ is an input combination of function (4.10).

To derive the weight of bit n, a pairwise comparison is performed between such tuples, and the difference

$$|f(g^{\boldsymbol{y}}_{n,1}) - f(g^{\boldsymbol{y}}_{n,0})| \tag{4.11}$$

is computed for all $\boldsymbol{y} \in \mathbb{B}^{s-1}$, where f is the significance-weighting function (4.1).

The final value of w_n is, then:

$$w_n = \max_{\boldsymbol{y}}(|f(g^{\boldsymbol{y}}_{n,1}) - f(g^{\boldsymbol{y}}_{n,0})|) \tag{4.12}$$

Figure 4.6 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.

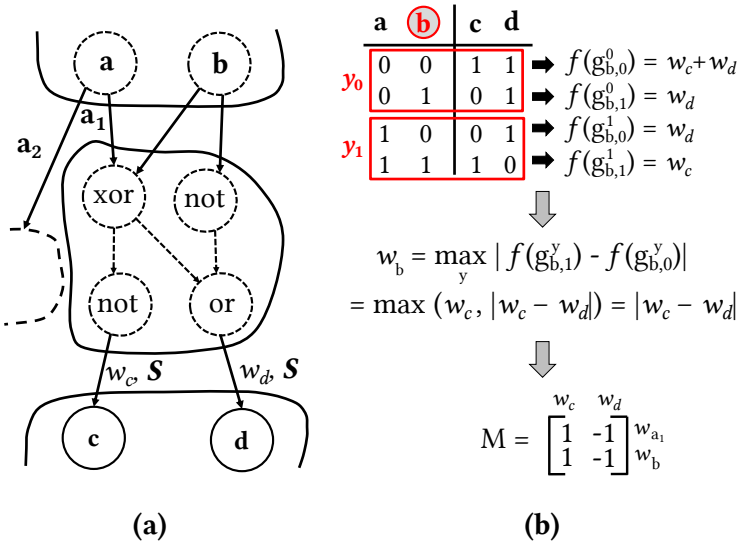


Figure 4.6. Propagation matrix derivation for an example subgraph. The figure reports its truth table, and differences are computed for w_b. The process is repeated twice (once for each input) to obtain the complete matrix.

In particular, Figure 4.6b illustrates how w_b is obtained: a pairwise comparison is performed between input tuples that differ only for the value of bit b. In this simple 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 (4.11) is equal to $|w_c - w_d|$. 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*). One 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.

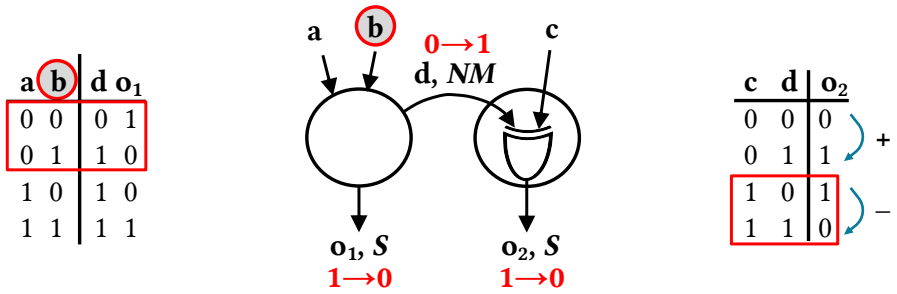
Weight w_b is then expressed as a linear combination of output weights w_c and w_d. The same process is repeated for input a_1, and the results are stored in a *propagation matrix* $M(|I_s|, |O_s|)$, where the i-th row contains the coefficients (0, 1 or -1) of the linear combination of the output weights for input i. Weights of the subgraph inputs can then be calculated through propagation function:

$$p(\ell, g) := M(g)\ell \tag{4.13}$$

where ℓ is the vector containing all labels of the subgraph output.

Propagation matrices are derived for each subgraph, then used to propagate subgraph output weights to their inputs. Indeed, the propagation function p for PP-Mon is the product between propagation matrix M and the vector of the subgraph output weights.

4.2.2 Non-monotonic outputs



(a) (b) (c)

Figure 4.7. An example of reverted propagation in non-monotonic output bits. A two-subgraph partition (b), with the truth table of the right subgraph (c) and that of the left one (a).

If, on the contrary, $f(\boldsymbol{o})$ is not monotonically increasing with subgraph output bits, the direction of subgraphs bit variations could be reverted in lower computations and, hence, one is forced to set the input weight to the *sum* of all flipped output bit weights, using

$$w_n = \max_y(|f(g^y_{n,1}) + f(g^y_{n,0})|)$$

instead of Equation (4.12).

Figure 4.7 illustrates an example of this behaviour: Fig. 4.7b reports 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, it is clear from the truth table of

the left subgraph (Fig. 4.7a) that the first comparison, enclosed in the red rectangle, would give $|w_d - w_{o_1}|$. However, d is input to the left subgraph, whose truth table is depicted in Fig. 4.7c: 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 tagged with information on monotonicity (*strict monotonic*, *non-strict monotonic*, and *non-monotonic*).

4.2.3 Non-strict monotonic outputs

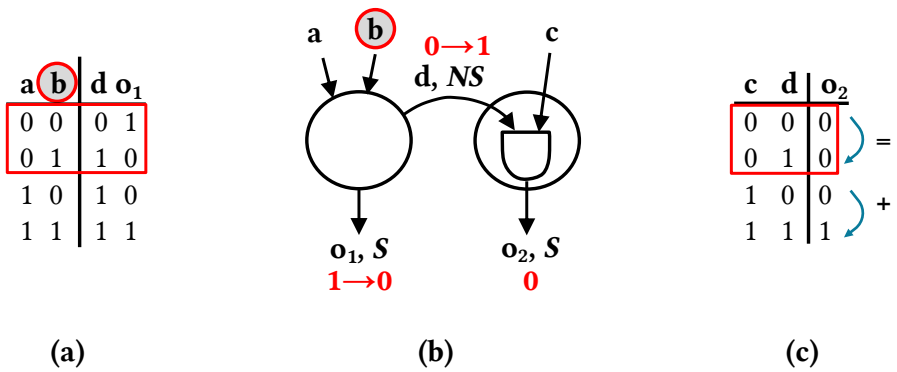


Figure 4.8. An example of propagation in non-strict monotonic output bits. A two-subgraph partition (b), with the truth table of the right subgraph (c) and that of the left one (a).

Finally, let us examine why it is important to distinguish between strict and non-strict monotonicity. Figure 4.8 illustrates a non-strict monotonic bit d: indeed, in Fig. 4.8c 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.

4.2.4 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 4.6), or children nodes are distributed in different subgraphs (as for node a in the same figure). 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, but applied to *each subgraph separately*, where the crucial difference with exhaustive simulation 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.

4.3 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. Therefore, a second alternative algorithm is proposed.

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:

$$I_i = [l_i, u_i] \tag{4.14}$$

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:

$$-I_i = I_{i,1\to 0} = [-u_i, -l_i] \tag{4.15}$$

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:

$$l_i = \min_{\boldsymbol{x}}(f(\boldsymbol{o}^{\boldsymbol{x}}_{i,1}) - f(\boldsymbol{o}^{\boldsymbol{x}}_{i,0})) \tag{4.16}$$

$$u_i = \max_{\boldsymbol{x}}(f(\boldsymbol{o}^{\boldsymbol{x}}_{i,1}) - f(\boldsymbol{o}^{\boldsymbol{x}}_{i,0})) \tag{4.17}$$

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:

$$w(n_i) = \max(|l_i|, |u_i|) \tag{4.18}$$

However, values of equations (4.16) and (4.17) are, in general, unknown. PP-Int aims to find *bounds* on such values, which will satisfy the following:

$$l_i \leq \min_{\boldsymbol{x}}(f(\boldsymbol{o}^{\boldsymbol{x}}_{i,1}) - f(\boldsymbol{o}^{\boldsymbol{x}}_{i,0})) \tag{4.19}$$

$$u_i \geq \max_{\boldsymbol{x}}(f(\boldsymbol{o}^{\boldsymbol{x}}_{i,1}) - f(\boldsymbol{o}^{\boldsymbol{x}}_{i,0})) \tag{4.20}$$

In other words, it aims to identify the *tightest* possible interval containing the actual one.

4.3.1 Primary output intervals

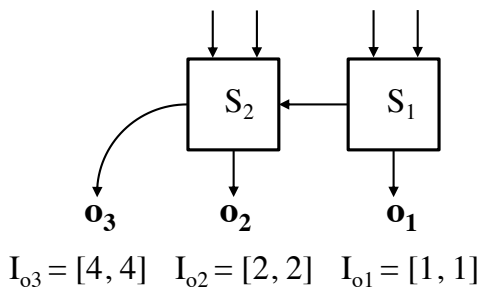


Figure 4.9. A simple example of a 2-bit ripple-carry adder, with intervals assigned to its primary outputs.

We have seen that a function $f : \mathbb{B}^k \rightarrow \mathbb{Z}$ 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.

As an example, Figure 4.9 illustrates 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 $I_{o_1} = [1,1]$.

4.3.2 Input interval derivation and propagation

Once primary output intervals are assigned, the partition graph is traversed bottom-up 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 4.10a, input c is inspected and these lines are enclosed in red rectangles.

As illustrated in Section 4.2, each truth table is represented 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. Function $g_{n,v}$ of Eq. (4.10) has been defined as the output of truth table g when its n-th input is forced to a constant value v.

Figures 4.10 and 4.11 illustrate interval propagation for input c in a generic subgraph (fig. 4.10a). 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 $I^{\boldsymbol{y}}$. Remember that the subgraph output labels are known, since propagation is performed bottom-up, starting from the circuit primary outputs: in this example, we assume that $I_d = [5,5]$ and $I_e = [0,2]$.

Figure 4.11 illustrates partial interval computations *per single value of* $\boldsymbol{y}$. For example, when $\boldsymbol{y} = [0\ 0]$, meaning that a and b are equal to 0, equation (4.10) provides values

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

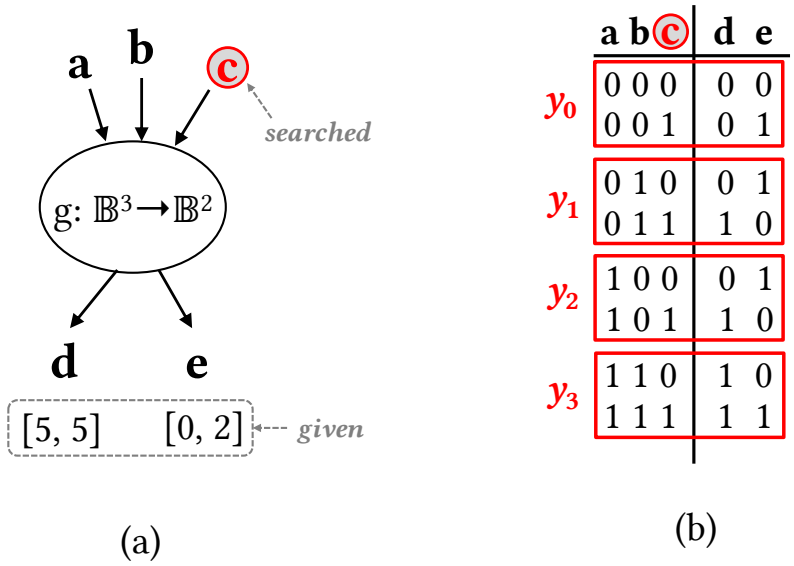


Figure 4.10. A generic subgraph (a) with output nodes labels given. Its truth table (b), along with the output labels, is used to compute intervals for its inputs.

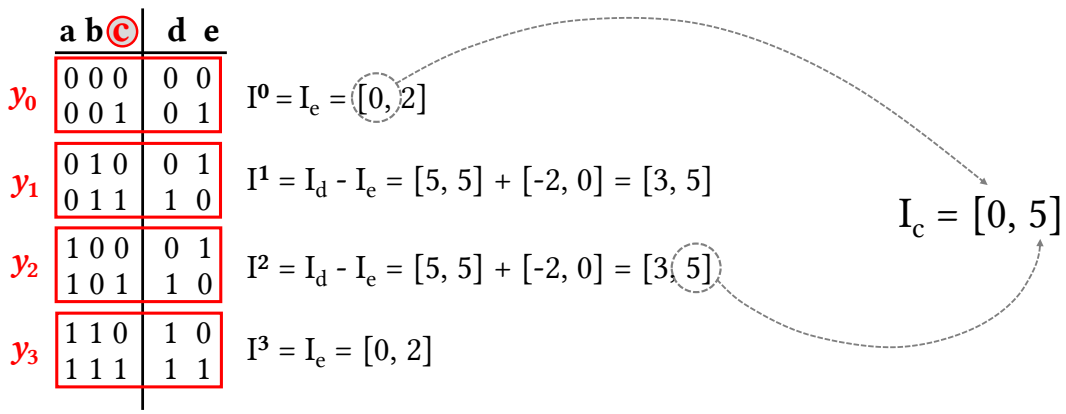


Figure 4.11. Detailed computation of intervals for bit c of Figure 4.10a.

which are the truth table outputs for input c fixed to 0 and to 1, respectively.

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

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

where, in this example, $I_e = [0, 2]$.

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

Therefore, the propagation function for input n is:

$$p(\ell, g)_n = \uplus_{y \in \mathbb{B}^{s-1}} \ell^T (g^y_{n,1} - g^y_{n,0}) \tag{4.21}$$

where $\uplus$ indicates interval union, denoting the smallest interval containing all argument intervals.

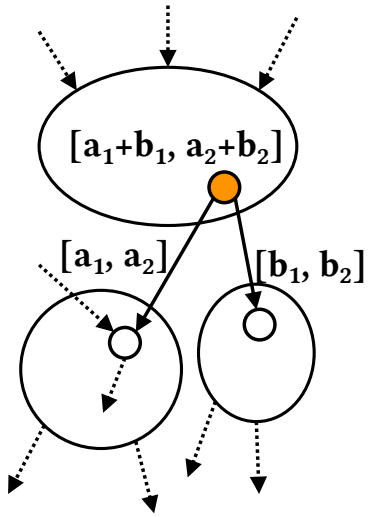


Figure 4.12. A node has two children in two different subgraphs, forcing the resulting interval to be set to the sum.

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, as in the example of Figure 4.12, its interval will be set to the sum of its children's intervals. Finally, the algorithm resorts 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 4.1 for rapid comparison.

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 information on lower bounds for

Table 4.1. Summary of the three algorithms subsumed by the Partition and Propagate framework.

method	**labels**	$p(\ell, g)$
sum propagation	$L = \mathbb{Z}$	$\sum_{i=1}^{t} \ell_i$
PP-Mon	$L = \mathbb{Z} \times \{S, NS, NM\}$	$M(g)\ell$
PP-Int	$L = \mathbb{R} \times \mathbb{R}$	$\uplus_{y \in \mathbb{B}^{s-1}} \ell^T (g^y_{n,1} - g^y_{n,0})$

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 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.

4.4 Sum propagation

Finally, it is useful to observe how an existing approach, sum propagation, fits in P&P formal framework. It is a very simple way of implementing a propagation-based error model, and it produces conservative bound estimations. Here, each node is considered individually and, hence, the graph is trivially partitioned so that each subgraph contains a single node.

Labels are positive integers, the weights, and if a node has label $\ell \in L$ it means that the absolute error is bounded by ℓ at that node, *i.e.* $\max_x |e_x| \leq \ell$.

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 example, in [1], and an example of graph labelled through sum propagation is given in Figure 4.13.

Explicitly,

$$p_j(\ell, g) := \sum_{i=1}^{t} \ell_i \tag{4.22}$$

for all inputs $1 \leq j \leq s$ of the considered node. Note that g, the function implemented by the sugbraph, and j, the subgraph input index, do not appear on the right-hand-side of the definition, leading to a fast but imprecise analysis.

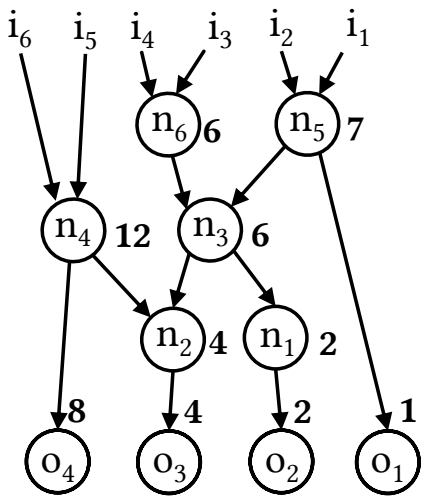


Figure 4.13. A DAG labelled with sum propagation.

4.5 Graph partitioning

Both PP-Mon and PP-Int employ the same partitioning strategy. Different partition choices for a given graph G will impact two aspects: the accuracy of the obtained weights on one side, and the feasibility of subgraph simulation on the other.

Figure 4.14 depicts two extreme partitions: at one end, each gate corresponds to one subgraph. This leads to a *feasible* partition (each subgraph has a limited number of inputs – namely two, at most, for up to two-inputs gates – and hence can be simulated) but will be the worst in terms of resulting weight accuracy, because it corresponds to resorting to the sum propagation algorithm. In fact, with this partition, all gate fanouts are external edges and the propagation step (Section 4.3.2) is conservatively forced to adopt sum propagation for all of them, when deriving weights of subgraph outputs.

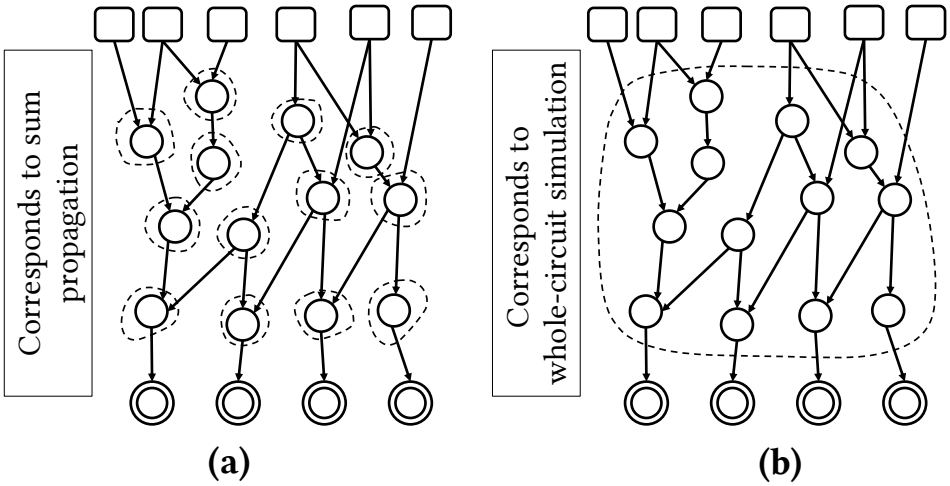


Figure 4.14. Two end-of-the-spectrum partitions: a) each gate corresponds to one subgraph, b) the whole circuit corresponds to a single subgraph.

At the other extreme, the whole circuit corresponds to a single subgraph. This will lead to the best partition in terms of resulting weight accuracy (all children of the same gate belong to the same sugraph, and hence get subsumed in a single propagation matrix), but will be generally infeasible, because it corresponds to whole-circuit simulation.

To find a suitable partition trading off these concerns, two main aspects must be considered. First, *feasibility*: the number of inputs $|I_s|$ for each subgraph s has to be small, in order to allow simulation of all the $2^{|I_s|}$ possible input combinations.

Secondly, as illustrated in Section 4.3.2, it is advantageous that *all children of any given node belong to the same subgraph*, so that gate fanouts are included into the same subcircuit simulation and sum propagation employment is unnecessary.

The partitioning algorithm proposed labels graph nodes with subgraph IDs, by first assigning the same subgraph ID to all children of the same node, iteratively merging subsets with same ID to 1) honour the condition for all nodes, and 2) guarantee an acyclic partition. In a second traverse, subgraphs for which $|I_{s+t}| \leq \max\{|I_s|, |I_t|\}$ are merged together, *i.e.*, the number of inputs of the resulting subgraph does not increase with respect to the largest one among its components. This is done to reduce the number of propagation matrices to be derived, without increasing the computational cost for any of them.

This algorithm creates a first phase partition, which is exemplified in Figure 4.15a. However, a partition thus created does not guarantee feasibility, be-

cause not all subgraphs necessarily will have a limited number of inputs (the feasible number of inputs is set to 3 for the simple example in the figure, and an infeasible subgraph exists at the top-left). Hence, to recover the feasibility property, those subgraphs that are infeasible are further partitioned using the one-node-one-subgraph extreme partition scheme, as exemplified in Figure 4.15b.

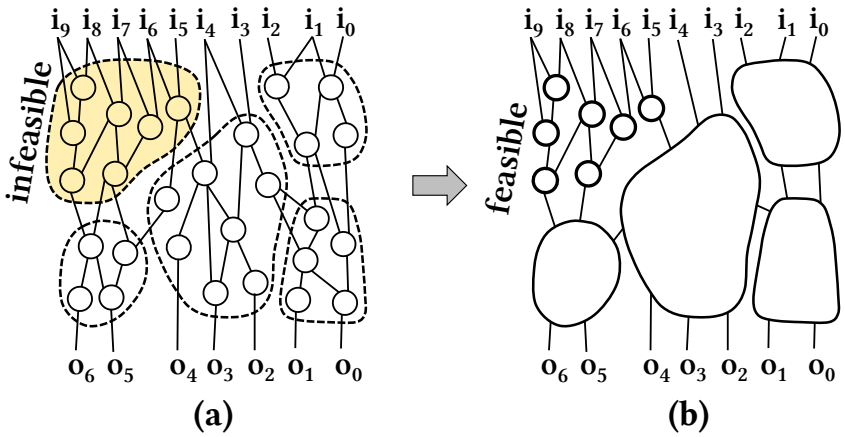


Figure 4.15. a) A first phase partition with one infeasible subgraph (top-left), b) A second phase feasible partition, with the infeasible subgraph further partitioned into single-gate subgraphs.

Thus, this two-stage partitioning strategy generates 1) a *feasible* partition, where 2) a high number of node fanouts are included in the same subgraph, and hence are subsumed in a single propagation matrix.

4.5.1 Time complexity analysis

For the sake of readability, in this section the cardinality of a set is indicated with the name of the set itself.

For a graph with N nodes, partition p is obtained through a first graph traverse, of cost N, for children assignment to a specific subgraph. The resulting partition is then explored to guarantee that, if there exist subgraphs leading to cycles, these are merged in a single one. This step has worst case complexity $\mathcal{O}(S + n_{EE})$, where S is the number of resulting subgraphs *before* further partitioning due to infeasibility, and n_{EE} the number of external edges. After the second subgraph-merging traverse of cost S, infeasible subgraphs are split into single-gate subgraphs in another linear step. Therefore, partitioning has complexity $\mathcal{O}(N + S + n_{EE})$.

Propagation, then, implies subgraph simulation for all subgraph inputs, and for all internal gates. Time complexity of this phase strongly depends on the partition: for the extreme partition of Figure 4.14b, it would be $\mathcal{O}(N2^I)$, while for that of Figure 4.14a only $\mathcal{O}(4N) = \mathcal{O}(N)$, since all gates have (at most) 2 inputs and, hence, 4 possible input patterns. For generic partitions, the time complexity is $\mathcal{O}(\sum_{s=1}^{S} N_s 2^{I_s})$. Since we bound I_s to a threshold T, the exponential term can be expressed as a constant $C = 2^T$, leading to $\mathcal{O}(C \sum_{s=1}^{S} N_s) = \mathcal{O}(CN) = \mathcal{O}(N)$.

In conclusion, the overall time complexity of both PP-Mon and PP-Int is $\mathcal{O}(N + S + n_{EE})$, which depends on the chosen partition parameters but is still remarkably lower than approaches that resort to simulation of all inputs or SAT-solver based exact weight derivation. In the experiments of the next section, C is set to 2^{10}, while the number of input combination for processed circuits ranges between 2^{16} and 2^{130}. With a Monte Carlo selection of a subset of possible input patterns the resulting simulation has a complexity of $\mathcal{O}(MN)$, given M the cardinality of such subset. However, to obtain accurate estimations, M must be considerably larger than C (for example, 2^{17} in [74], 2^{20} in [4] and more than 2^{22} in [1]).

4.6 Performance of the proposed algorithms

To assess the performance of my error modeling algorithms over the state of the art, I have implemented the three error propagation techniques (sum, PP-Mon and PP-Int) over a wide set of benchmark circuits specified in VHDL, described in Table 4.2. All circuits were synthesized with Synopsys Design Compiler, targeting a 40nm technology library, while exhaustive simulation and subgraph simulation was performed with SIS [66].

Table 4.2. 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
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
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
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

4.6.1 P&P-Intervals values I_i

As described in Section 4.3, PP-Int assigns to each circuit gate a label I_i, with a lower bound l_i and an upper bound u_i. The corresponding weight is derived by taking the maximum absolute value of the two, as in eq. (4.18). Figure 4.16 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 4.17, ADDER8 weights coincide with the simulated ones, while ABS_DIFF weights are, generally, more distant.

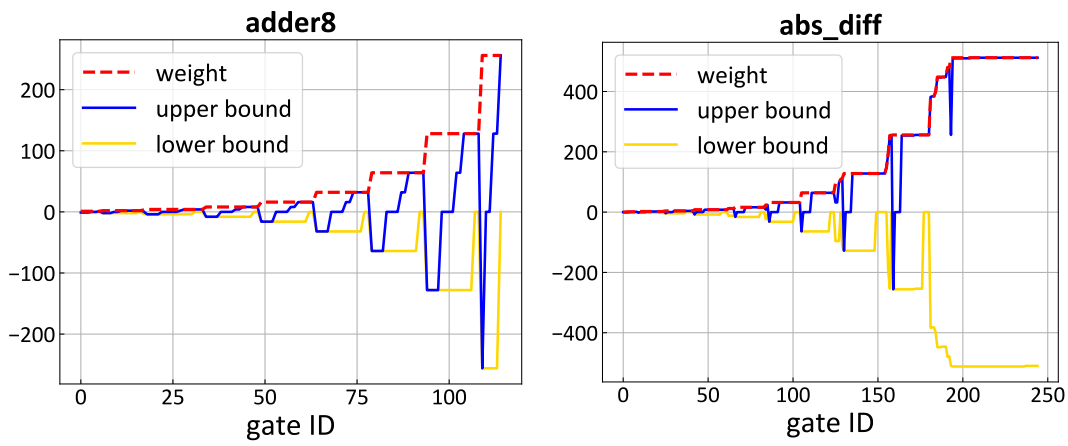


Figure 4.16. Lower and upper bounds found by PP-Int, and the corresponding weight, for adder8 (left) and abs_diff (right).

4.6.2 Weight values comparison

Figure 4.17 compares weights obtained by PP-Int, derived from the corresponding interval, and by PP-Mon, against the state-of-the-art strategy sum propagation. 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, otherwise by increasing PP-Int weight. We can observe that PP-Mon and PP-Int weights are very close (or even equal) to the simulated ones, and 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). This showcases the effectiveness of the proposed P&P framework in improving error modeling accuracy.

We can also remark that PP-Mon and PP-Int weights are almost always identical, except for MULT8 benchmark and for a very small set of gates in ABS_DIFF and BUTTFLY. Indeed, 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.

Larger benchmarks have also been considered to assess the efficiency of the proposed approach. Figure 4.18 reports six large benchmarks, whose average gate count is $\approx$ 2400 (with a maximum of 5344 for the DIST circuit).

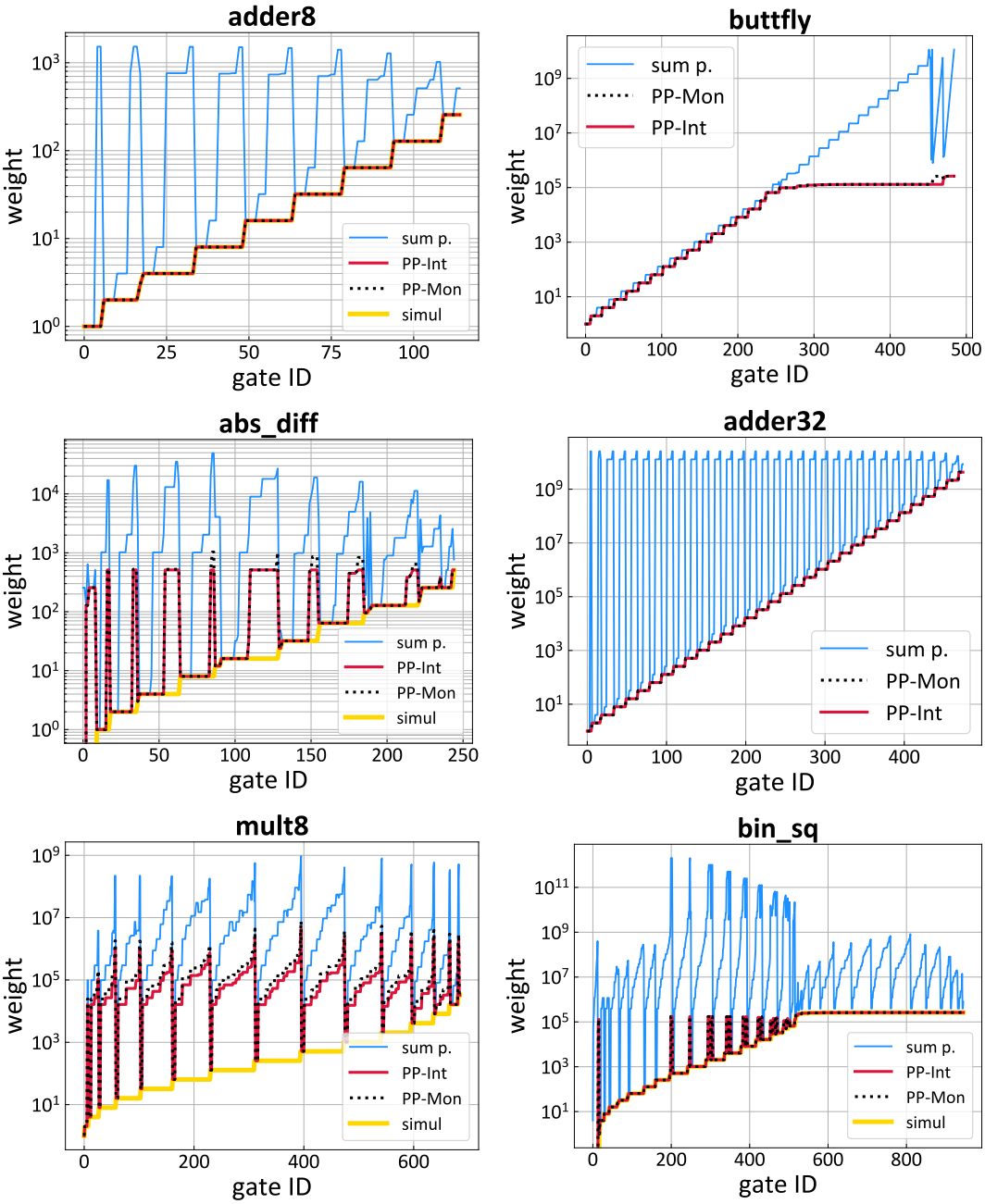


Figure 4.17. Comparison of weights obtained through sum propagation, PP-Mon and PP-Int. For the same gate (identified by a gate ID), the weights obtained by the three algorithms are reported, along with the simulated values (when available). Gates are sorted by increasing (simulated or PP-Int) value.

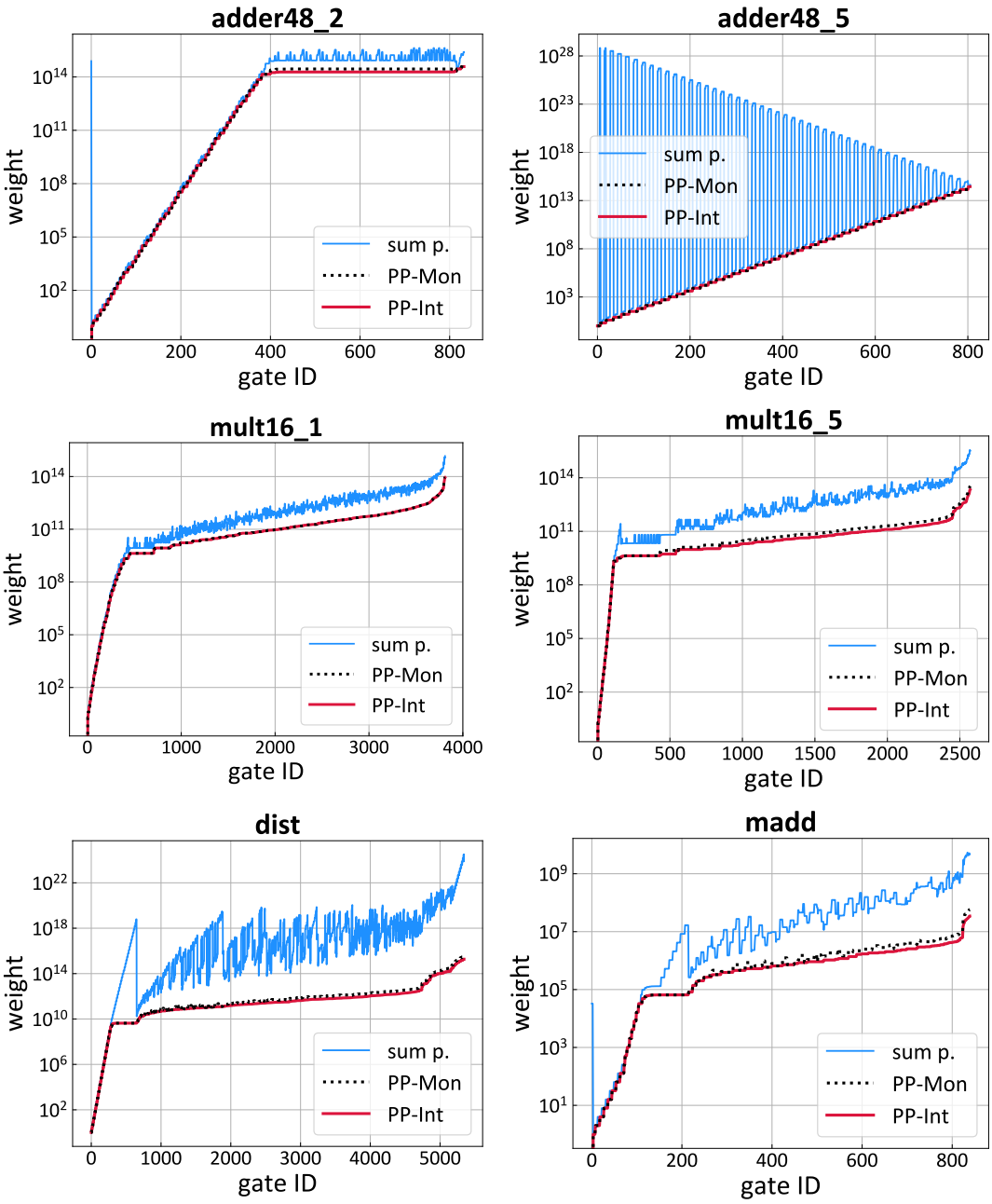


Figure 4.18. Comparison of weights obtained through sum propagation, PP-Mon and PP-Int for adders (top), multipliers (center), euclidean distance and multiply-add (bottom).

The top row of Figure 4.18 illustrates 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 4.6.4.

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 interval error formulation.

4.6.3 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 significance, and functioning in MSD-first fashion [75, 76]. In these adders, each input bit x_i corresponds to a pair of bits, x_i^+ and x_i^-, selected such that $x_i = x_i^+ - x_i^-$ [75]. 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 a partition that isolates full-adder-like blocks, but with different subgraph functionalities. Figure 4.19 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.

4.6.4 Faster vs slower circuits

Finally, it is interesting to further study weights accuracy for circuits with the same functionality but with different topological structure, imposed by different delay constraints at synthesis time.

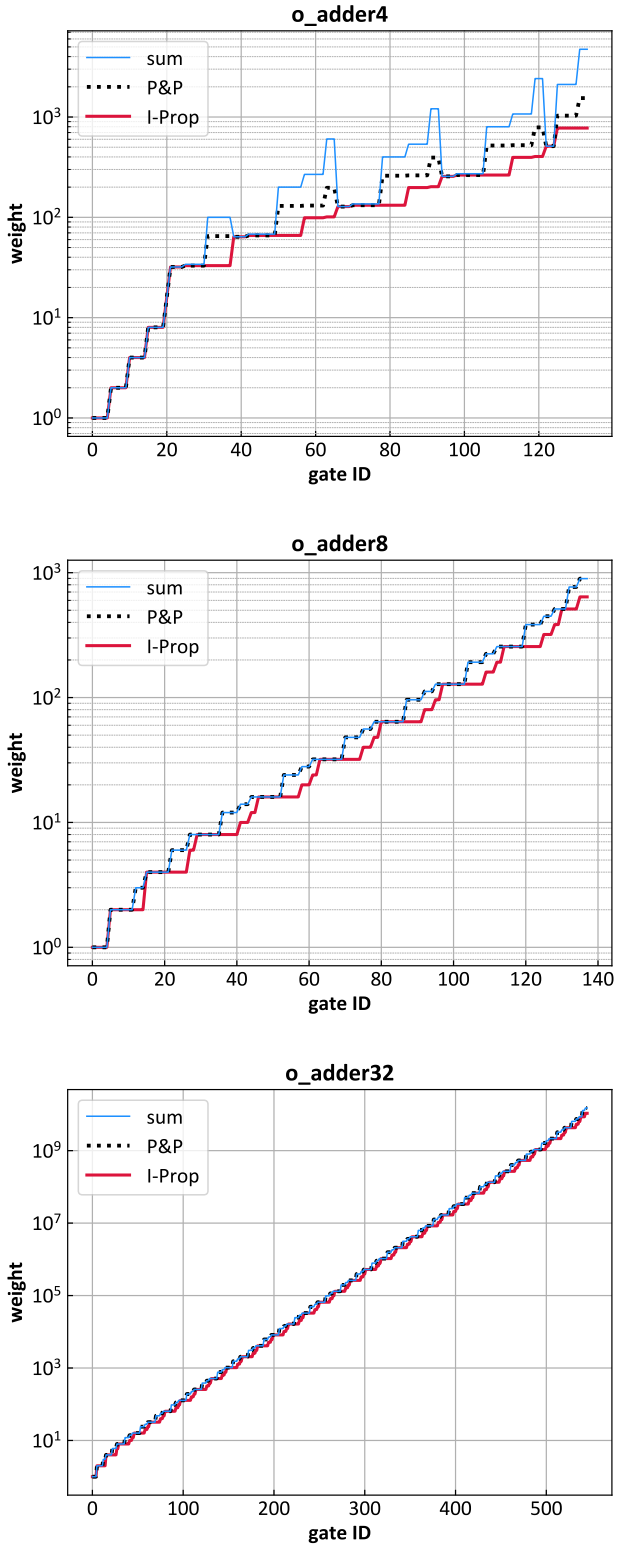


Figure 4.19. Comparison of weights obtained through sum propagation, PP-Mon and PP-Int for online adders.

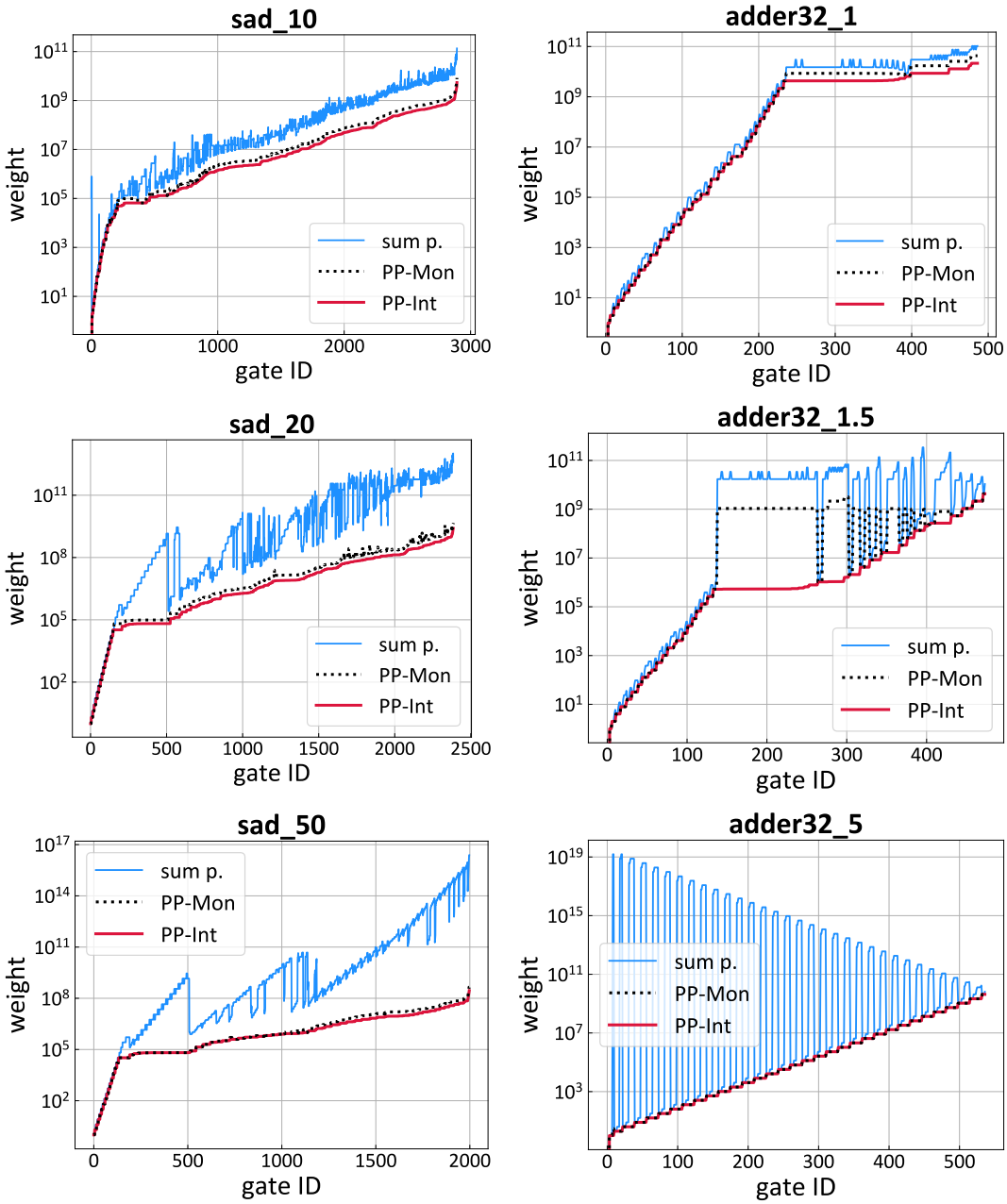


Figure 4.20. Comparison of weights obtained through sum propagation, PP-Mon and PP-Int for SAD with different delay constraints (left column) and 32-bit adders (right column).

Figure 4.20 showcases weights retrieved for three 16-bit SAD circuits (left

column) and three 32-bit adders (right column). Their delay constraint increases top to bottom, 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, it can be seen 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, it can be concluded that the efficiency of PP-Int and PP-Mon can vary significantly according to the circuit structure, but these methods always perform orders of magnitude better than sum propagation and, in many cases, PP-Int improves even considerably the accuracy of weights obtained through PP-Mon.

4.6.5 Maximum number of inputs per subgraph

Table 4.3. Influence of threshold T on resulting partitions, in terms of number of resulting subgraphs, infeasible subgraph ratio, and average distance from exact weights (when available).

T	ADDER8			ABS_DIFF			ADDER32			BUTTFLY			MULT8			BIN_SQ		
	n. sg.	**inf. %**	**avg dist.**	**n. sg.**	**inf. %**	**avg dist.**	**n. sg.**	**inf. %**	**avg dist.**	**n. sg.**	**inf. %**	**avg dist.**	**n. sg.**	**inf. %**	**avg dist.**	**n. sg.**	**inf. %**	**avg dist.**
10	7	0.0	0.0	17	0.0	1.4e2	31	0.0	-	15	0.0	-	190	9.6	2.3e5	8	0.0	4.4e3
5	7	0.0	0.0	135	44.1	2.3e3	31	0.0	-	15	0.0	-	190	9.6	2.3e5	272	16.1	1.3e7
2	63	44.3	3.5e2	163	55.6	3.0e3	231	36.3	-	430	78.9	-	340	30.4	6.6e5	648	57.0	3.8e9

The partitioning strategy described in Section 4.5 allows to trade-off the accuracy of weights and computational effort by tuning, during the partitioning phase, threshold $T = \max(|I_s|)$, the maximum number of inputs allowed in a subgraph. Showcasing the impact of this parameter, Table 4.3 reports the number of obtained subgraphs, the infeasible subgraph ratio (*i.e.*, the number of gates assigned to infeasible subgraphs over the total number of gates), and the average distance from exact weight, for $T = 10$, $T = 5$ and $T = 2$ in all benchmarks of Table 4.2. As can be expected, lower thresholds result in more and smaller subgraphs, and consequently in more conservative weights, as more subgraphs

are split into one-node-one-subgraph partitions. A value of $T = 10$ obtains high quality weights, while the resulting computation time remains within seconds to minutes, and it is the value chosen for all experiments previously reported in this section.

Figure 4.21 provides a graphical representation of the above-mentioned phenomenon: the graph reports weights obtained by sum, PP-Mon with $T = 5$, PP-Mon with $T = 10$, and the simulated weights for ABS_DIFF. It is evident that lowering threshold T results in a weights curve (pink) approaching the sum propagation one (blue).

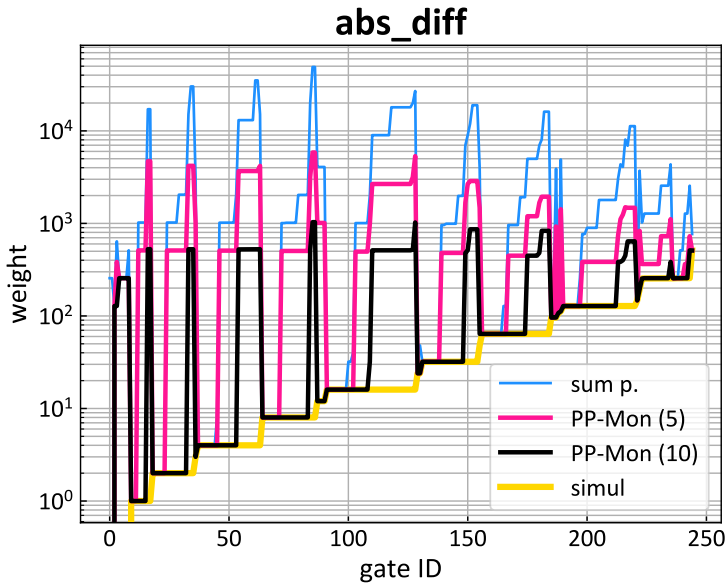


Figure 4.21. When the number of maximum allowed inputs per subgraph is reduced, PP-Mon weights approach sum propagation weights.

4.7 Effectiveness of error modeling for ALS

The improved accuracy in gate weights achieved by PP-Int and PP-Mon has a significantly positive impact on the subsequent simplification of inexact circuits. To demonstrate this effect, the GLP [1] approximation strategy described in Section 2.1.1 was reimplemented in two versions. GLP iteratively selects gates to be removed until the error constraint is violated, starting from gates of lower weight. In the original implementation, weights are assigned with sum propagation algorithm, while in the new version these weights are derived through

PP-Mon.

Figure 4.22 compares the reduction in Energy, Delay and Area (EDAP) for the two versions of GLP, with varying error constraints. The figure illustrates how the PP-Mon algorithm guides GLP synthesis to high-performance approximate circuits, resulting in large EDAP gains for small error thresholds.

Indeed, apart from the BUTTFLY benchmark, for which the two methods obtain exactly the same approximate circuits, the gap between the two curves is extremely wide, especially for larger circuits.

In conclusion, the experimental results illustrated in this section confirm the validity of both novel methods for error modeling, P&P-Intervals and P&P-Monotonicity, which highly improve the weight accuracy when compared with the state of the art. Moreover, accurate weights prove to be very effective in guiding ALS algorithms towards highly efficient solutions, further legitimating the research effort spent on error modeling for Approximate Logic Synthesis.

P&P-Monotonicity was published in a full paper at the Design and Automation Conference (DAC) in 2019 [5]; the paper also contains the graph partitioning strategy described in this chapter. P&P-Intervals, instead, is described in a journal paper currently under review at the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD).

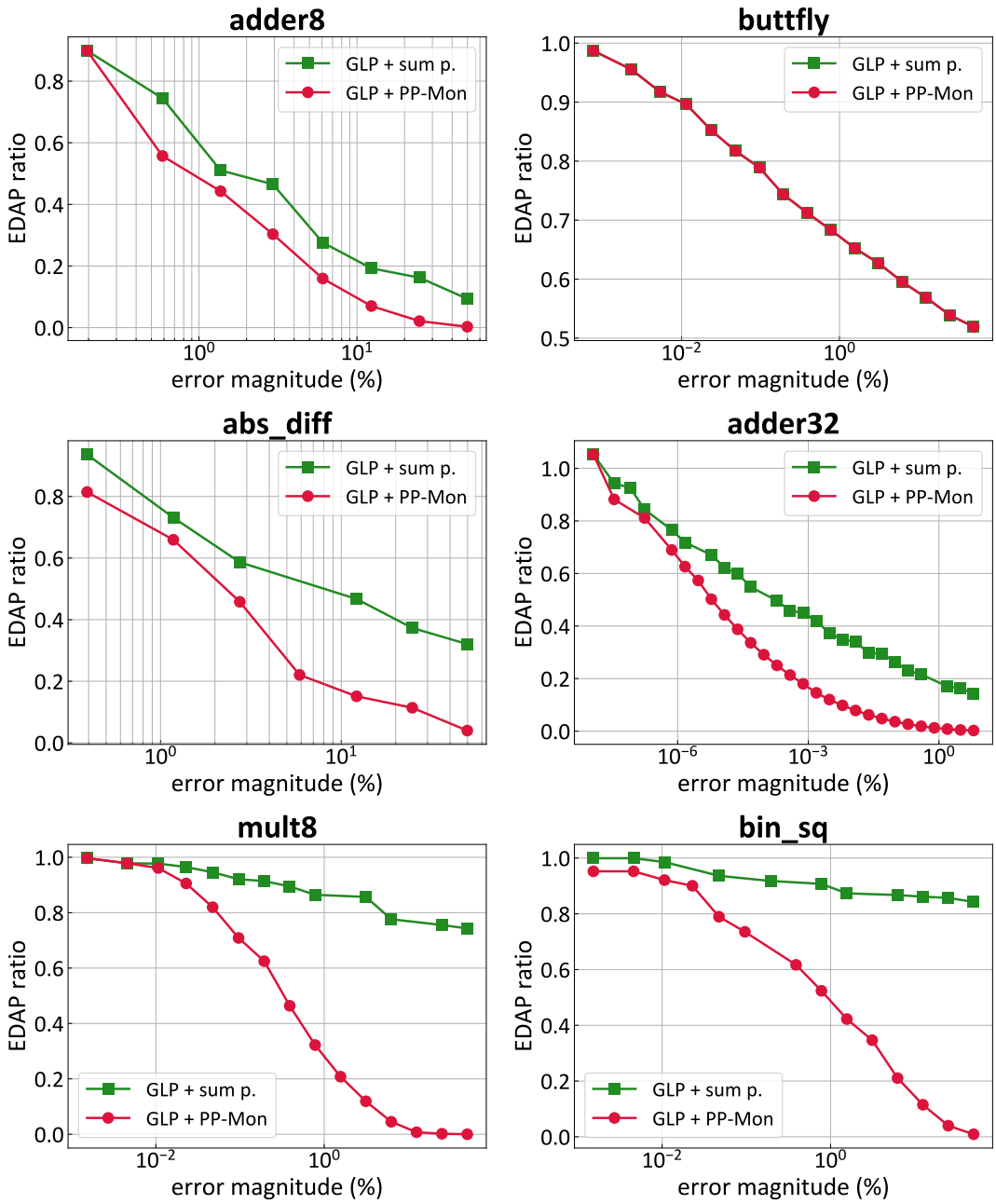


Figure 4.22. Energy-Delay-Area Product (EDAP) of the inexact circuits derived from GLP [1] with its original sum propagation error modeling, and guided by weights derived through PP-Mon [5]. EDAP values are normalized with respect to the exact implementations.

Chapter 5

Dealing with expected error

During my research activity, a considerable effort has been dedicated to adapting the formal framework described in Chapter 4, born and tailored to guarantee *bounds* on maximum error, to provide *estimates* on the expected error. Research in this direction is still ongoing: taking into account error distribution, as opposed to extreme values alone, introduces a whole new set of research questions.

However, I believe in the importance of enriching the existing framework, since many ALS algorithms in the state of the art focus on controlling this error metric instead of maximum error. Despite the fact that problems arise, that were not manifest for maximum error, the research activity carried on so far still suggests promising results.

5.1 Problem definition

Once again, the aim is to derive a gate-level error model of the circuit; this time, though, providing estimates on the expected error that can be seen on the circuit primary output if a gate is removed. Figures 4.3 and 4.4 of the previous chapter are reported here (Figure 5.1 and 5.2) for the sake of readability. Referring to Figure 5.1, which depicts three circuits that are identical except for node n_i, consider the expected value of the absolute difference between exact and approximate output:

$$\mathbb{E}\{|e_{\boldsymbol{x}}|\} = \mathbb{E}\{|f(\boldsymbol{o}^{\boldsymbol{x}}) - f(\boldsymbol{o}^{\boldsymbol{x}}_{i,0})|\} \tag{5.1}$$

Remember that $\boldsymbol{o}^{\boldsymbol{x}}$ is the Boolean output of the exact circuit for input $\boldsymbol{x}$, while $\boldsymbol{o}^{\boldsymbol{x}}_{i,0}$ is the Boolean output of the approximate circuit where node i is set

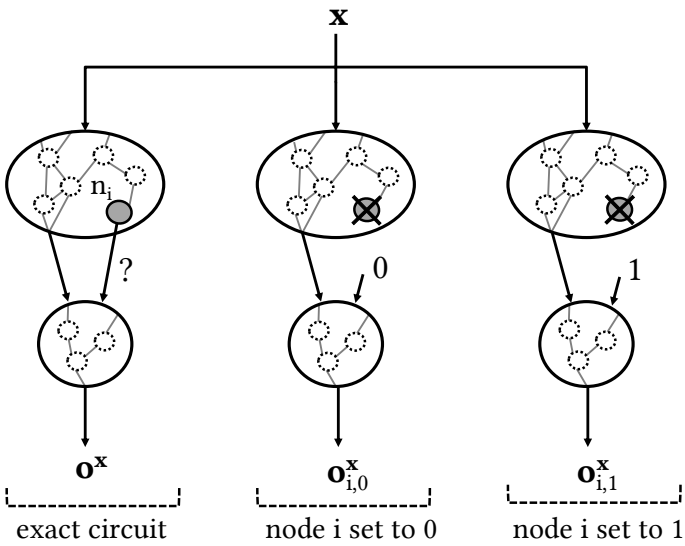


Figure 5.1. comparison between exact circuit and approximate versions with a gate set to 0 or 1.

to 0, for the same input $\boldsymbol{x}$. Function f maps the Boolean output into an integer (for example, bit-significance weighting). Given error

$$e_{01,\boldsymbol{x}} = f(\boldsymbol{o}^{\boldsymbol{x}}_{i,1}) - f(\boldsymbol{o}^{\boldsymbol{x}}_{i,0}) \tag{5.2}$$

it still holds that

$$|e_{\boldsymbol{x}}| \leq |e_{01,\boldsymbol{x}}| \tag{5.3}$$

and, hence

$$\mathbb{E}\{|e_{\boldsymbol{x}}|\} \leq \mathbb{E}\{|e_{01,\boldsymbol{x}}|\} \tag{5.4}$$

However, it is clear that to obtain a good estimate for Eq. (5.1) it is necessary to estimate the probability of node n_i being 0 (and 1).

Graph (N, E) representing the circuit is again partitioned with the same technique described in Section 4.5, and Figure 5.2 represents a generic subgraph implementing a function g with s inputs and t outputs.

To derive the s input labels, the propagation function p this time will need the t output labels $l \in L$, function $g : \mathbb{B}^s \to \mathbb{B}^t$, and the s probabilities $P \in \mathbb{R}$ of *input* nodes being 0:

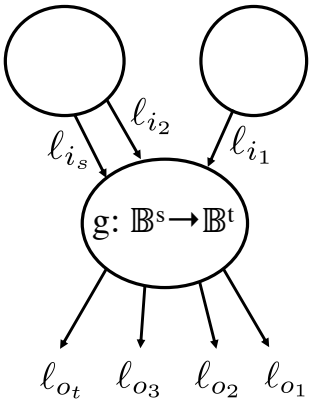


Figure 5.2. Label propagation model for a generic subgraph with s inputs and t outputs, implementing function g.

$$p : L^t \times (\mathbb{B}^s \times \mathbb{B}^t) \times \mathbb{R}^s \longrightarrow L^s \tag{5.5}$$

Indeed, when computing labels for the inputs of a subgraph, errors are weighted by the probability of having that specific input combination. The mechanism to derive such labels will be explained in the next sections; for the moment, it is important to note that there can be two possible strategies for expected error modeling:

1. propagating maximum errors, and weighting these errors by the probability of having the corresponding input in each subgraph;

2. propagating average errors, always weighting them by the probability of having the corresponding input.

In both cases, labels are expressed in the interval form of PP-Int (Section 4.3), since it has proven to be more precise in weight computation and easier to implement.

Once the graph is partitioned, it is possible to estimate the probabilities $P(n_i = 0)$ for all subgraph output nodes. Indeed, given a primary input probability distribution, it is possible to simulate all subgraphs separately, in a top-down graph traverse, and observe their truth tables. For instance, consider again the subgraph implementing function g in Figure 5.2. The probability of node o_1 of being 0 can be estimated with

$$P(o_1 = 0) = \frac{|\{x \mid g(\boldsymbol{x})_{o_1} = 0\}|}{2^s} \tag{5.6}$$

in other words, the number of times node o_1 is zero over the number of possible outputs of the truth table, *where the inputs are assumed to be i.i.d.*

5.2 Maximum error propagation

In this method, there are two different labels for each subgraph nodes: labels in L obtained through PP-Int algorithm, and labels in $\overline{L}$ obtained by weighting the former by the corresponding input probabilities.

Consider now a truth table similar to that of section 4.3.2, implementing function $g : \mathbb{B}^3 \rightarrow \mathbb{B}^2$ and depicted in Figure 5.3. In particular, the figure illustrates how interval label I_c is computed through a pairwise comparison between tuples that differ only for value of bit c. The final label for c is the smallest interval containing all intermediate intervals, hence $I_c = [-5, 7]$.

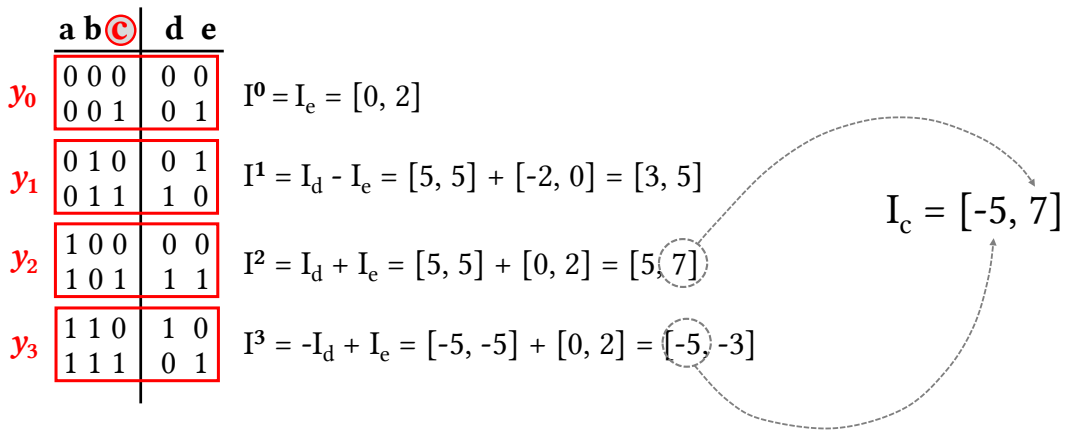


Figure 5.3. Detailed computation of intervals for bit c, where $I_{d,0\rightarrow 1} = [5,5]$ and $I_{e,0\rightarrow 1} = [0,2]$ are given.

As already stated in Section 4.3, a weight can be associated to an interval $I_n = [l_n, u_n]$ by taking the largest absolute value of its extremes. Let us call the function w:

$$w(I_n) = w_n = \max(|l_n|, |u_n|)$$

thus, $w(I_c) = 7$. Now, remember that

$$g^{x}_{n,v} = g(x_1, ..., x_{n-1}, v, x_n, ..., x_{s-1}).$$

represents the output of truth table g when its n-th input is forced to a constant value v. It has one fewer input than g, and $\boldsymbol{y} \in \mathbb{B}^{s-1}$ is one of its input combinations. In maximum error propagation, the expected weight $\overline{w}_n$ is obtained by

$$\overline{w}_n = p(\ell, g, P)_n = \sum_{\boldsymbol{y} \in \mathbb{B}^{s-1}} w(\ell^T(g^{y}_{n,1} - g^{y}_{n,0}))P(\boldsymbol{y}) \tag{5.7}$$

In the example of Figure 5.3, the above expression, where labels l are intervals, translates to

$$\overline{w}_c = w(I_0)P(a, b = 0, 0) + w(I_1)P(a, b = 0, 1) + w(I_2)P(a, b = 1, 0) + w(I_3)P(a, b = 1, 1)$$

An important assumption at this point is that input distributions are independent and, therefore, $P(a, b = x, y) = P(a = x)P(b = y)$. For this simple example, we can assume all four possible values of $\boldsymbol{y}$ are equally likely, hence leading to

$$\overline{w}_c = 2 \cdot \frac{1}{4} + 5 \cdot \frac{1}{4} + 7 \cdot \frac{1}{4} + 5 \cdot \frac{1}{4} = 4.75$$

The rationale behind this strategy is that PP-Int labels keep track of the actual error values, ideally producing very tight bounds, and that these values are then weighted by the probability of their occurrence.

5.3 Average error propagation

A valid alternative to the previous methodology is to have a single label in $\overline{L}$ for each gate value, such that

$$\overline{L} = \overline{I}_n = [\mathbb{E}\{l_n\}, \mathbb{E}\{u_n\}] = [\overline{l}_n, \overline{u}_n] \tag{5.8}$$

which is coupled to the expected weight

$$\overline{w}_n = \max(|\bar{l}_n|, |\overline{u}_n|) \tag{5.9}$$

In this case, primary output intervals would be set to their bit significance times the probability of encountering an error (hence, the probability of that primary input being 1 in the exact circuit). These intervals would be then propagated upwards by taking the average values of their extremes, and weight $\overline{w}_n$ would be

$$p(\ell, g, P)_n = w(\sum_{\boldsymbol{y} \in \mathbb{B}^{s-1}} \ell^T (g^{\boldsymbol{y}}_{n,1} - g^{\boldsymbol{y}}_{n,0}) P(\boldsymbol{y})) \tag{5.10}$$

Again, in the example of Figure 5.3, this means

$$\overline{w}_c = w(I_0 P(a, b = 0, 0) + I_1 P(a, b = 0, 1) + I_2 P(a, b = 1, 0) + I_3 P(a, b = 1, 1))$$

that, with the same assumption of input independence, leads to

$$\overline{w}_c = \max((0 + 3 + 5 - 5) \cdot \frac{1}{4}, (2 + 5 + 7 - 3) \cdot \frac{1}{4}) = \max(0.75, 2.75) = 2.75$$

As expected, weights obtained with this strategy are lower than those obtained through maximum propagation.

5.4 Preliminary results and comparisons

Figure 5.4 showcases a comparison between the expected weights $\overline{w}_n$ obtained by the different strategies of method 1 (maximum error propagation), method 2 (average error propagation), and Monte Carlo random input selection for ADDER8. Indeed, Monte Carlo simulation represents the obvious baseline comparison for methods dealing with expected error estimation. The graph reports two Monte Carlo curves, one obtained by simulating the 0.006% of the total number of inputs, and the other with the 0.012%. Finally, exact values of the average error are reported in the green curve (full simulation).

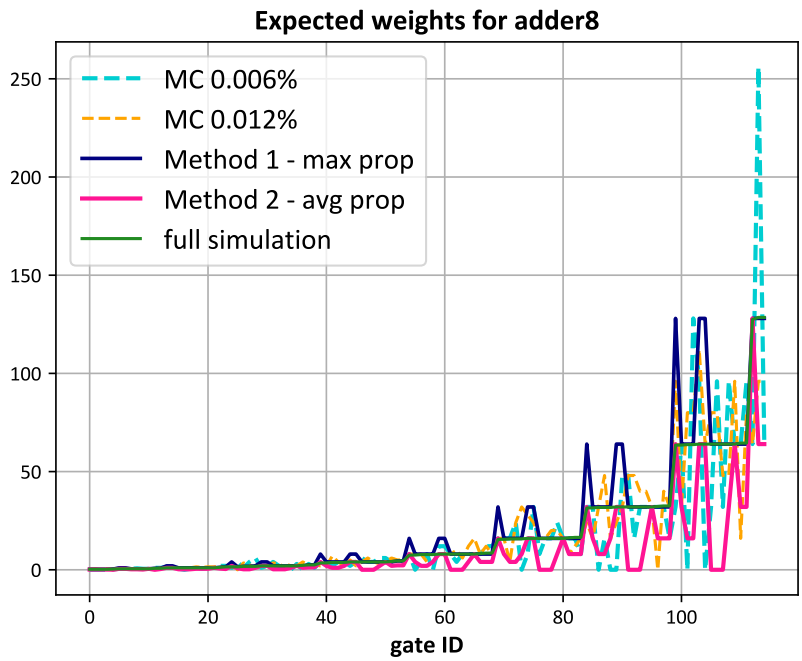


Figure 5.4. Expected weights for adder8, comparison between the two proposed methods and Monte Carlo random input selection.

Table 5.1. Accuracy of the proposed methods against Monte Carlo Simulation on adder8.

method	**ARE %**	**AAE**
MC 0.006%	41.7	8.73
MC 0.012%	33.1	6.3
Method 1 - max prop	21.3	3.45
Method 2 - avg prop	49.2	8.88

Table 5.1 reports, for each curve of Figure 5.4, the average relative error (ARE) and average absolute error (AAE) on all circuit gates. As introduced in Chapter 2, the relative error for each gate is computed as

$$\frac{|\overline{w}_{n,method} - \overline{w}_{n,exact}|}{\overline{w}_{n,exact}} * 100$$

The absolute error, instead, is simply the numerator of the fraction in the expression above. These quantities are then averaged on all $|N|$ circuit nodes.

Method 1, maximum error propagation, systematically overestimates the average weights, although its accuracy is the best of all four methods: indeed, its ARE% and AAE are the smallest. On the contrary, method 2, average error propagation, not only underestimates weights, but its accuracy appears to be the worst of all four methods. These results do not reflect the accuracy of the worst-case error version of P&P since, for ADDER8 benchmark, both PP-Mon and PP-Int retrieved exact weights.

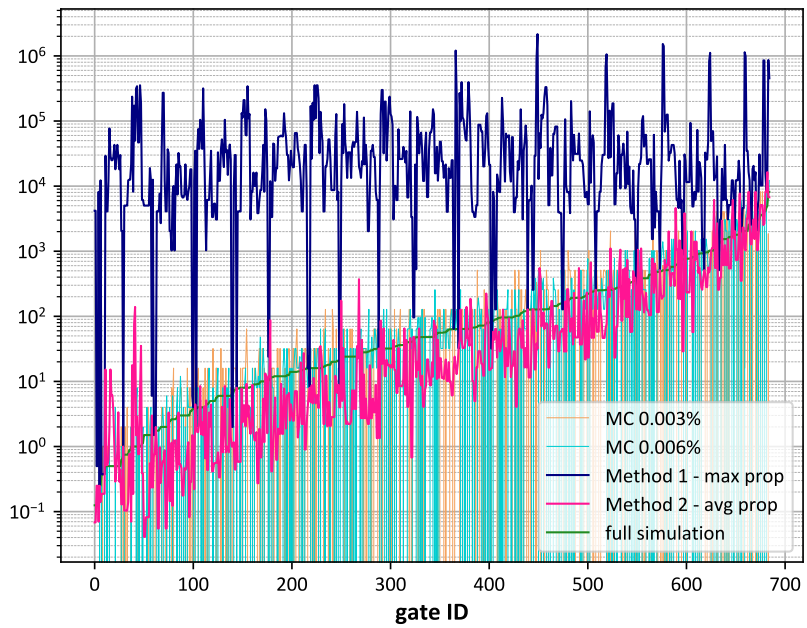


Figure 5.5. Expected weights for mult8, comparison between the two proposed methods and Monte Carlo random input selection.

Figure 5.5 reports the same curves for benchmark MULT8, whose worst-case weights were orders of magnitude larger than the real ones. Here, the results seen so far are reversed: method 1 strongly overestimates weights $\overline{w}_n$, due to the scarce accuracy of PP-Int intervals. Method 2, on the other hand, retrieves weights closer to the actual values, with less outliers than Monte Carlo simulation. These results are confirmed in Table 5.2, which again quantifies the performance of these methods: error metrics for method 1 show that the intervals are simply too wide to provide a reasonable estimate. Method 2, instead, presents

Table 5.2. Accuracy of the proposed methods against Monte Carlo Simulation on mult8.

method	**ARE %**	**AAE**
MC 0.003%	111.9	342.3
MC 0.006%	82.0	251.7
Method 1 - max prop	6.5e4	6.6e5
Method 2 - avg prop	125.5	227.6

the lowest AAE of all four methods, although its ARE is beaten by both Monte Carlo simulations.

In conclusion, both methods envisaged so far to adapt my formal framework to expected error do not seem to be highly competitive with Monte Carlo simulation, although these approaches are much faster than simulation with a large number of inputs.

The scarce accuracy of methods 1 and 2 may depend on the erroneous assumption of subgraph inputs independence: indeed, independence of inputs in a subgraph as the one described by the truth table of Figure 5.3 may be a too strong assumption, since their values are determined by the circuit logic above.

Moreover, in Method 1, sometimes intervals are too conservative to be exploited for average error estimation, since their worst-case values are too far from the average ones. A potential strategy could be the study of the error probability distribution over each interval, but this might lead to unfeasible computational complexity due to the multiple propagation steps.

In any case, the derivation of global properties from local observation has proven not to be trivial for average-case error, and leads to many challenging research questions for the future.

Chapter 6

Achievements and future work

6.1 Publications

During my PhD I have published the following peer-reviewed articles:

- Ilaria Scarabottolo, Giovanni Ansaloni, George A. Constantinides and Sherief Reda, "Approximate Logic Synthesis: A Survey", Proceedings of the IEEE Journal, Approximate Computing Special Issue, 2020.
- Giovanni Ansaloni, Ilaria Scarabottolo and Laura Pozzi. "Judiciously Spreading Approximation among Arithmetic Components with Top-Down Inexact Hardware Design", Applied Reconfigurable Computing, Toledo, 2020.
- Ilaria Scarabottolo, Giovanni Ansaloni, George A. Constantinides and Laura Pozzi. "Partition and Propagate: an Error Derivation Algorithm for the Design of Approximate Circuits". Design Automation Conference, 2019.
- Ilaria Scarabottolo, Giovanni Ansaloni and Laura Pozzi. "Work-in-Progress: A Partitioning Strategy for exploring Error-Resilience in Circuits". Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, 2018.
- Ilaria Scarabottolo, Giovanni Ansaloni and Laura Pozzi. "Circuit Carving: A Methodology for the Design of Approximate Hardware". Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2018.
- Ilaria Scarabottolo, Cesare Alippi and Manuel Roveri. "A spectrum-based adaptive sampling algorithm for smart sensing", 2017 IEEE SmartWorld, San Francisco, CA, 2017.

Moreover, the following paper is currently under review:

- Ilaria Scarabottolo, Giovanni Ansaloni, George A. Constantinides and Laura Pozzi. "A Formal Framework for Error Estimation in Approximate Logic Synthesis". Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2020.

6.2 Ongoing research

After my graduation, I will continue my research activity at USI as a Post-Doc researcher, in Professor Laura Pozzi's research group.

The results of my research work can be of great benefit to a vast range of applications where energy efficiency is crucial, ranging from high performance computing to ultra-low-power scenarios. This is supported, and encouraged, by the great interest that both research and industry have shown for Approximate Computing, which is now largely recognised as a valid solution for energy-hungry applications.

In particular, my work implements hardware optimisations, which have shown to be a valid and efficient way to obtain remarkable energy savings that could not be obtained by software approximations alone. Indeed, the combination and smooth coupling of approximate computing techniques both at software and hardware level is a flourishing research topic [8]. Moreover, since I am working on an automatic framework, I embed the advantages of software optimisations in hardware: usability, versatility and transparency to the user (architect or programmer).

I am planning to extend my work in several directions in the upcoming years, as illustrated in the next sections.

6.2.1 Collaborations with Prof. S. Reda

My survey on Approximate Logic Synthesis [77] was written with Professor S. Reda from Brown University, Rhode Island, USA. This first collaboration was very profitable, since we have exchanged points of view on our research subjects and we have compared strengths and weaknesses of our approaches.

In particular, we have appreciated the good performance of one of his ALS algorithms, BLASYS [4], described in Section 2.1.2. Approximate circuits obtained through BLASYS present very large area reductions, in exchange for small errors in the output quality. However, to identify which subcircuit is most suited for

approximation, they employ a highly time-consuming Monte Carlo evaluation of the induced error. Our first idea has been to substitute this Monte Carlo step with our Interval Propagation algorithm, so to have a fast and accurate indication on subcircuits most amenable to approximation.

Moreover, we are planning to organize a tutorial on Approximate Logic Synthesis, where we would show how all different techniques we have developed or surveyed operate, and observe the differences in performance. To do so, we have already shared an open-source benchmark repository, BACS, available on Github. [1]

I am convinced that this collaboration may lead to many other interesting works in ALS in the future.

6.2.2 Exploiting the primary inputs characteristics

At present, no specific assumption is made on the circuits primary inputs when performing error modeling, apart from considering them independent and identically distributed. We can imagine two different directions for exploiting the primary input properties more in depth, in order to further improve the labelling accuracy.

Firstly, we could investigate which input subsets maximise the error obtained for each subgraph. Indeed, input subsets that lead to the worst-case error for a given subgraph may have much lower impact on other subgraphs on the path to the primary outputs. If this was the case, one could take advantage of such property to further improve weight accuracy, possibly reducing the final weight of some gates.

A second attractive research direction is related to the deterministic *vs* stochastic data representation [78]. While current formulations assume a deterministic, noise-free representation of data, dealing with noise-affected data would imply significant modifications in the error propagation model. For instance, our circuit primary inputs may be data acquired from a sensor, where the value of some bits may hold *with a certain probability*. Uncertainty propagation in noise-affected data is a widely studied topic [78], and it would be interesting to study how this effect combines with my error propagation model, where intrinsic errors and induced errors interact in the synthesis of an approximate circuit.

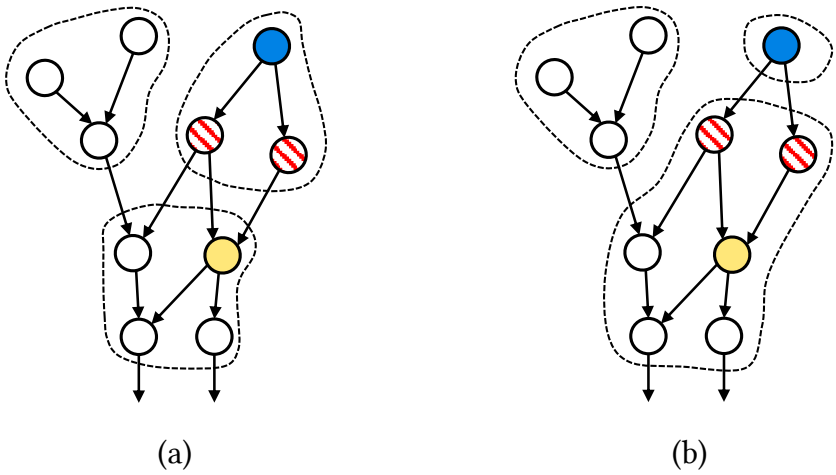


Figure 6.1. (a) A circuit where partitioning breaks the reconvergent path between the blue and yellow nodes, inserting intermediate computations at the striped red nodes and (b) a circuit where the reconvergent path is fully included into the same subgraph.

6.2.3 Re-thinking partitioning

Another important extension of my current work consists in modifying the partitioning algorithm of Section 4.5, in order to improve weight accuracy of PP-Int or PP-Mon. In fact, we have remarked that breaking reconvergent paths between nodes in partitioning often leads to weights overestimation.

This phenomenon is exemplified in Figure 6.1: indeed, two distinct paths originate from the blue, filled node and reconverge at the lower yellow node. If the graph is partitioned as in Figure 6.1a, these paths are cut and nodes split into different subgraphs. When performing weight propagation, both PP-Int and PP-Mon will compute a weight for the red striped nodes in the first pass, when external edges are considered. This means that their weights will be employed to compute that of the blue node, possibly inducing overestimation, since the effect of their bitflips will be considered separately, as if their fanouts where completely distinct. Instead, the yellow node value is determined exactly by that of the striped ones: its weight will be considered twice, instead of once, while computing the weight of the blue one.

Figure 6.1b shows a possible solution to this problem: the two striped nodes have been included into the lower subgraph and, hence, the weight of the blue node will be computed with all nodes on the reconvergent paths subsumed into

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

the same subgraph, thus canceling the overestimation effect.

I plan to define and study a *reconvergence graph*, where an edge exist between node a and node b if, in the original graph, there are two – or more – reconvergent paths between such nodes. The great challenge will then be to understand how to distribute nodes and realise feasible subgraphs, since reconvergence inclusion may lead to very large subgraphs, which could not be simulated. I believe that this approach could strongly impact on the overall weight accuracy and, hence, on the usability of my algorithms for ALS.

6.2.4 Approximate Neural Networks

Another interesting extension of my current work is the application of approximate arithmetic circuits to larger scale applications. An example is to employ approximate units in Neural Network layers, as done in [79–82].

Hanif *et al.* [79] have proposed a way to determine which layers of a CNN are most resilient and, hence, can be approximated more aggressively. It would be interesting to compare my results with this work in two different directions:

1. apply Circuit Carving to adders and multipliers of more resilient layers; and

2. employ my error modeling framework at coarser grain, between different arithmetic circuits within the same layer, to identify which one should be approximated first.

Regarding the second point, it would become necessary to re-define the function that assigns primary output weights, since they would not be simple binary numbers. An example of alternative function could be the probability of obtaining the correct classification at the final output.

6.2.5 Circuit Carving reformulation: heuristics for scalability

Circuit Carving suffers from its worst-case exponential formulation, which prevents it from finding good approximations of large circuits in a reasonable amount of time. A possible alternative to this approach would be to re-define Circuit Carving exploration, so that its exhaustive tree search is substituted by a heuristics.

More in detail, after having labelled all nodes with their weights, it could be possible to add them to the cut, starting from those of lower weights. However, as opposed to the GLP approach, in-loop Monte Carlo simulation would not be necessary: one could simply use the sum of the outgoing edges as a good estimate

of the cut influence, as Circuit Carving already employs accurate weights. In practice, this would represent exploring *a single branch* in the search tree, instead of navigating it all. The resulting approach would be much faster than the current version of CC – and of GLP as well, hence representing a promising alternative to the current state of the art.

6.3 Conclusion

My PhD research work has greatly evolved during the past four years, starting from the field exploration, then developing effective ALS methods, to finally end up with the definition of the Partition and Propagate framework. As any academic reader will know, such a journey necessarily addresses only a small subset of all interesting research questions that arise along the way, opening infinitely many doors and never really closing any of them. Indeed, plenty of exciting challenges remain open and, while this thesis represents a landmark in my exploration of Approximate Computing, it is far from being the last one.

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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, Jun 2018.

[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*, June 2019.

[6] 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.

[7] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys (CSUR)*, vol. 48, p. 62, May 2016.

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

[9] A. Sinha, A. Wang, and A. P. Chandrakasan, "Algorithmic transforms for efficient energy scalable computation," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 31–36, July 2000.

[10] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pp. 124–134, Sept. 2011.

[11] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pp. 25–34, 2010.

[12] M. Samadi, J. Lee, D. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," *MICRO 2013 - Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 12 2013.

[13] M. Samadi, D. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," vol. 49, pp. 35–50, 02 2014.

[14] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *SIGPLAN Notices*, vol. 47, pp. 301–312, Mar. 2012.

[15] U. R. Karpuzcu, I. Akturk, and N. S. Kim, "Accordion: Toward soft near-threshold voltage computing," in *Proceedings of the 20th International Symposium on High-Performance Computer Architecture*, pp. 72–83, Feb 2014.

[16] D. J. Palframan, N. S. Kim, and M. H. Lipasti, "Precision-aware soft error protection for GPUs," in *Proceedings of the 20th International Symposium on High-Performance Computer Architecture*, pp. 49–59, Feb 2014.

[17] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM refresh-power through critical data partitioning," *ACM SIGPLAN Notices*, vol. 46, pp. 213–224, Mar 2011.

[18] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *MICRO 46: Proceedings of the 46st Annual International Symposium on Microarchitecture*, MICRO-46, pp. 25–36, ACM, Dec 2013.

[19] 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.

[20] 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.

[21] K. Shi, D. Boland, E. Stott, S. Bayliss, and G. A. Constantinides, "Datapath synthesis for overclocking: Online arithmetic for latency-accuracy trade-offs," in *Proceedings of the Design Automation Conference 2014*, ACM, 2014.

[22] K. E. Murray, A. Suardi, V. Betz, and G. Constantinides, "Calculated Risks: Quantifying Timing Error Probability with Extended Static Timing Analysis," *IEEE Trans. Computer-Aided Design*, vol. 38, pp. 719–732, Mar. 2018.

[23] 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.

[24] 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.

[25] M. Shafique, W. Ahmad, R. Hafiz, and J. Henkel, "A low latency generic accuracy configurable adder," in *Proceedings of the 52nd Design Automation Conference*, pp. 86:1–86:6, ACM, 2015.

[26] N. Zhu, W. L. Goh, W. Zhang, K. S. Yeo, and Z. H. Kong, "Design of low-power high-speed truncation-error-tolerant adder and its application in digital signal processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 8, pp. 1225–1229, 2010.

[27] H. A. F. Almurib, T. N. Kumar, and F. Lombardi, "Inexact designs for approximate low power addition by cell replacement," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 660–665, 2016.

[28] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, (New York, NY, USA), p. 820–825, 2012.

[29] Z. Yang, A. Jain, J. Liang, J. Han, and F. Lombardi, "Approximate xor/xnor-based adders for inexact computing," in *2013 13th IEEE International Conference on Nanotechnology (IEEE-NANO 2013)*, pp. 690–693, 2013.

[30] 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.

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

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

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

[34] W. Liu, L. Qian, C. Wang, H. Jiang, J. Han, and F. Lombardi, "Design of approximate radix-4 booth multipliers for error-tolerant computing," *IEEE Transactions on Computers*, vol. 66, no. 8, pp. 1435–1441, 2017.

[35] S. Ullah, S. S. Murthy, and A. Kumar, "Smapproxlib: Library of fpga-based approximate multipliers," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2018.

[36] H. Saadat, H. Javaid, and S. Parameswaran, "Approximate integer and floating-point dividers with near-zero error bias," in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 161:1–161:6, ACM, June 2019.

[37] S. Hashemi, R. I. Bahar, and S. Reda, "A low-power dynamic divider for approximate applications," in *IEEE/ACM Design Automation Conference*, no. 105, 2016.

[38] L. Chen, J. Han, W. Liu, and F. Lombardi, "On the design of approximate restoring dividers for error-tolerant applications," *IEEE Transactions on Computers*, vol. 65, no. 8, pp. 2522–2533, 2016.

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

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

[41] S. Venkataramani, V. Kozhikkottu, A. Sabne, K. Roy, and A. Raghunathan, "Logic synthesis of approximate circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2019.

[42] 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.

[43] K. Nepal, Y. Li, R. Bahar, and S. Reda, "Abacus: A technique for automated behavioral synthesis of approximate computing circuits," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, p. 361, Mar. 2014.

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

[45] H. Jiang, C. Liu, L. Liu, F. Lombardi, and J. Han, "A review, classification, and comparative evaluation of approximate arithmetic circuits," vol. 13, Aug. 2017.

[46] Y. Wu, C. Shen, Y. Jia, and W. Qian, "Approximate logic synthesis for fpga by wire removal and local function change," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 163–169, 2017.

[47] S. Sinha and W. Zhang, "Low-power fpga design using memoization-based approximate computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 8, pp. 2665–2678, 2016.

[48] 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.

[49] 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.

[50] Z. Peng, X. Chen, C. Xu, N. Jing, X. Liang, C. Lu, and L. Jiang, "Axnet: Approximate computing using an end-to-end trainable neural network," *CoRR*, vol. abs/1807.10458, 2018.

[51] E. Wang, J. J. Davis, R. Zhao, H.-C. Ng, X. Niu, W. Luk, P. Y. K. Cheung, and G. A. Constantinides, "Deep neural network approximation for custom hardware," *ACM Computing Surveys*, vol. 52, p. 1–39, May 2019.

[52] G. Ansaloni, I. Scarabottolo, and L. Pozzi, "Judiciously spreading approximation among arithmetic components with top-down inexact hardware design," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2020.

[53] 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.

[54] Y. Wu, Y. Li, X. Ge, Y. Gao, and W. Qian, "An efficient method for calculating the error statistics of block-based approximate adders," *IEEE Transactions on Computers*, vol. 68, no. 1, pp. 21–38, 2019.

[55] C. Li, W. Luo, S. S. Sapatnekar, and J. Hu, "Joint precision optimization and high level synthesis for approximate computing," in *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, (New York, NY, USA), Association for Computing Machinery, 2015.

[56] W. J. Chan, A. B. Kahng, S. Kang, R. Kumar, and J. Sartori, "Statistical analysis and modeling for error composition in approximate computation circuits," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pp. 47–53, 2013.

[57] M. Shafique, R. Hafiz, S. Rehman, W. El-Harouni, and J. Henkel, "Invited - cross-layer approximate computing: From logic to architectures," in *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, 2016.

[58] A. Agrawal, J. Choi, K. Gopalakrishnan, S. Gupta, R. Nair, J. Oh, D. A. Prener, S. Shukla, V. Srinivasan, and Z. Sura, "Approximate computing: Challenges and opportunities," in *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pp. 1–8, 2016.

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

[60] D. Shin and S. K. Gupta, "A new circuit simplification method for error tolerant applications," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 1–6, Mar. 2011.

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

[62] Z. Vasicek and L. Sekanina, "Evolutionary approach to approximate digital circuits design," *IEEE Transactions on Evolutionary Computation*, vol. 19, pp. 432–444, July 2015.

[63] 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.

[64] 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.

[65] Y. Wu and W. Qian, "An efficient method for multi-level approximate logic synthesis under error rate constraint," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, June 2016.

[66] 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.

[67] 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*, p. 364, Mar. 2014.

[68] S. Hashemi and S. Reda, "Generalized matrix factorization techniques for approximate logic synthesis," in *ACM/IEEE Design Automation and Test in Europe*, pp. 1289–1292, 2019.

[69] J. Ma, S. Hashemi, and S. Reda, "Approximate Logic Synthesis Using BLASYS," in *Workshop on Open-Source EDA Technology*, no. 5, 2019.

[70] A. Lingamneni, C. Enz, K. Palem, and C. Piguet, "Synthesizing parsimonious inexact circuits through probabilistic design techniques," vol. 12, p. 93, May 2013.

[71] Y. Yao, S. Huang, C. Wang, Y. Wu, and W. Qian, "Approximate disjoint bi-decomposition and its application to approximate logic synthesis," in *2017 IEEE International Conference on Computer Design (ICCD)*, pp. 517–524, Nov. 2017.

[72] M. Soeken, D. Große, A. Chandrasekharan, and R. Drechsler, "BDD minimization for approximate computing," pp. 474–479, Jan. 2016.

[73] 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.

[74] 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, Jun 2018.

[75] 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.

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

[77] I. Scarabottolo, G. Ansaloni, G. A. Constantinides, L. Pozzi, and S. Reda, "Approximate logic synthesys: A survey," *Proceedings of the IEEE journal, special issue on Approximate Computing*, Jul 2020.

[78] C. Alippi, *Intelligence for Embedded Systems*. Switzerland: Springer International Publishing, 1st ed., 2014.

[79] M. A. Hanif, R. Hafiz, and M. Shafique, "Error resilience analysis for systematically employing approximate computing in convolutional neural networks," in *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, pp. 913–916, IEEE, 2018.

[80] H. Tann, S. Hashemi, and S. Reda, *Lightweight Deep Neural Network Accelerators Using Approximate SW/HW Techniques: Methodologies and CAD*, pp. 289–305. 01 2019.

[81] M. Hanif, M. U. Javed, R. Hafiz, S. Rehman, and M. Shafique, *Hardware–Software Approximations for Deep Neural Networks: Methodologies and CAD*, pp. 269–288. 01 2019.

[82] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu, "Approxann: An approximate computing framework for artificial neural network," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 701–706, 2015.