Inferential Logic: A Machine Learning Inspired Paradigm for Combinational Circuits

Inferential Logic: a Machine Learning Inspired Paradigm for Combinational Circuits

Valerio Tenace, Andrea Calimera
Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino (Italy)

***Abstract*—Machine learning (ML) theories and tools suggest alternative forms to conceive and represent relationships among data. The same theories find their application in the Boolean domain, where logic functions can be described as inference rules. This paper introduces *Inferential Logic*, a novel paradigm that leverages the ML concept of statistical inference for the design of combinational logic circuits, the *Inferential Logic Circuits* (ILCs). The new design concept is conceived for low-power circuits that run *quasi-exact* computation in error-resilient applications, but it also provides an *exact* run-mode that can be dynamically enabled when accuracy scaling is not an option.**

I. Introduction

Machine learning (ML) is a paradigm in which hardware or software systems replicate a few simple learning/reasoning mechanisms proper of the human brain to resolve complex relationships among data [1]. The strength of ML tools lies under their ubiquity; they orthogonally apply to problems of different nature using statistical models on the collected data, just as the brain does through inductive reasoning over previous acquired knowledge.

Although the first evidence of such techniques dates back to the mid-19th century, recent advancements in computer architectures and massively parallel computing have prompted renewed interest on the matter. ML is inspiring new research trends which might have huge impact on many commercial and scientific fields. The Electronic Design Automation (EDA) is not an exception. The EDA community is mainly focusing on efficient hardware mapping of brain-inspired computing models, e.g., deep learning, while little effort is being spent on investigating how to take advantage of biological mechanisms to solve EDA problems. In this context, most noticeable contributions include the works by Li Wang [2], [3], where ML techniques are shaped to address testing and verification of digital circuits by means of several supervised and unsupervised techniques. Inspired by a previous work by Guzey [4], Li Wang also describes the concept of a supervised Boolean function learning mechanism where a statistical model evaluates the output of the function in order to reconstruct the original logic. Apart from these examples, very few has been done in terms of design strategies; for instance, a proper flow to build logic circuits using ML is currently not available.

In our view, the potential of ML techniques should be exploited more efficiently in order to create logic circuits that mimic how the human brain works, namely, to implement inaccurate, yet fast logic reasoning [5], [6]. This paradigm shift encompasses the replacement of exact logic rules and Boolean operators in favor of statistical models and inference tools.

Moving towards this objective, in this paper we describe a new design methodology where the representation of a generic Boolean function is obtained by means of a learning problem. The goal is to represent the behavior of a logic function through a more compact abstract model that works as a statistical description of the function itself. Such description is then used to *infer* the outcomes of the Boolean function up to a certain level of accuracy. Once mapped on a piece of hardware, the resulting circuit runs the *quasi-exact* computation of the logic function; we called this block the Inference Unit. However, since for some applications accuracy degradation is not an option, we describe how the inferential model can be reinforced in order to reach the full description of the logic function, hence, the *exact* computation. In such a case the Inference Unit is supported by an auxiliary unit called Supervisor Unit. We refer to this new class of circuits as Inferential Logic Circuits (ILCs) and we provide a complete framework for their logic synthesis and optimization.

Experimental results collected from an ILC embedded into an error-resilient application demonstrate that quasi-exact computation achieves an average accuracy of 94%, yet using 22× less devices w.r.t. a circuit counterpart obtained with a state-of-the-art standard design flow. Moreover, we quantify the figures of merit of other ILC benchmarks thus to give a fair comparison against a multi-level logic optimizer that obeys Boolean rules.

II. Background

In the panorama of machine learning, Classification Trees (CTs) represent a class of methods for the construction of classification models. Solving a classification problem encompasses two main stages: the *training* stage, which aims at generating the statistical abstract representation of the model, i.e., the CT; the *validation* stage, which quantifies the level of accuracy of the trained CT.

The training stage makes use of a labeled data-set with n observations; each observation is described by p predictor variables $X = \{x_1, \dots x_p\}$ and is labeled by one of the m available classes $y_i \in Y = \{y_1, \dots y_m\}$. During the validation stage, the CT is used to classify never occurred samples described by X over the label-set Y; the accuracy is given by the number of new samples that are correctly classified.

Building a CT implies the search of a "good" partitioning of the input space [7]. A CT implements a recursive partitioning using a proper sequence of comparisons between predictor variables and numeric thresholds. Let us assume to have a training data-set of people living in a city described through several predictors, or features, such as X={*Age, Weight,*

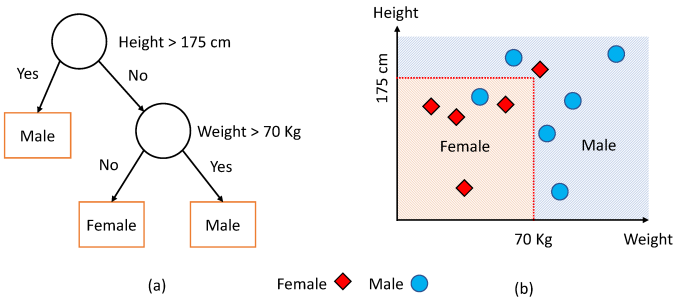

Figure 1. Visual description of a Classification Tree. Logical structure (left), domain partitioning (right).

Salary, Height, Nationality}. If the objective is to classify people by gender, i.e., $Y = \{Male, Female\}$, an abstract model with reasonably good accuracy would select which predictors are the most significant among the available predictors X in order to separate the training population into two different, and ideally disjoint clusters. This concept is graphically depicted in Figure 1, where the selected predictors are *Height* and *Weight*. The most important components learned during training are: the best list of split predictors, i.e., for each iteration which variable $x_k \in X$ is able to separate the training data-set into different groups with the lowest level of impurity [8], and the evaluation of the optimal threshold for each slipt. These two concepts are combined together into a statistical dispersion index called Gini index [8]. It is therefore possible to assume that a CT can be used to select which predictors better describe the training population.

Both predictor variables and labels can assume any type of value, e.g., *Weight* and *Height* are real numbers, *Male* or *Female* are categorical values; entirely-binary classification problems come with predictors and labels in the form of *binary* variables, i.e., *True* or *False*, 0 or 1.

III. Computing through Inference

A. Exact and Quasi-Exact Representation

A generic Boolean function $\mathcal{F}$ can be defined as an entirely-binary classification problem with a training set described by its truth table. The evaluation of an input pattern encompasses the classification on two possible labels: logic-0 or logic-1. As per the description given in Section II, a CT selects the most significant predictors, namely, the primary inputs that best describe the function $\mathcal{F}$. Since the selection is done through a statistical method, it is unlikely that a CT gives a complete cover of $\mathcal{F}$. In that sense, a CT is a *quasi-exact* description of $\mathcal{F}$. To notice that this is slightly different from the classical concept of approximate logic function. Indeed, the CT may cover both the ON-set and the OFF-set of $\mathcal{F}$.

In order to better understand this concept and describe how to achieve an *exact* representation, let us resort to a simple function $\mathcal{F} : \mathbb{B}^3 \rightarrow \mathbb{B}$ defined as in (1). It represents the function to be learned, and its ON-set is graphically depicted in Figure 2-a by means of a positional cube representation.

$$\mathcal{F}(a,b,c) = (a \wedge b) \vee (a \wedge c) \vee \neg a \tag{1}$$

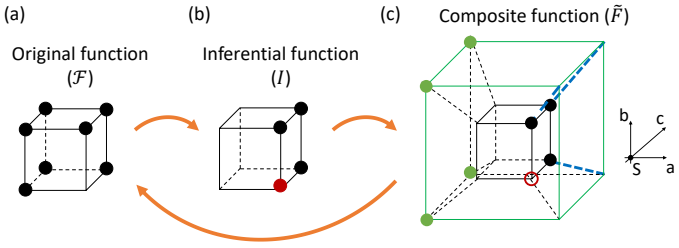

Figure 2. Positional cube notation of the example functions.

Let us then consider a function $\mathcal{I} : \mathbb{B}^3 \rightarrow \mathbb{B}$ defined as in (2), being its cube representation reported in Figure 2-b.

$$\mathcal{I}(a,b,c) = a \tag{2}$$

Assume that $\mathcal{I}$ is used to mimic $\mathcal{F}$: three over eight minterms are covered, i.e., $\mathcal{I} \equiv \mathcal{F}$ for three input patterns over eight. $\mathcal{I}$ covers a sub-set of the ON-set of $\mathcal{F}$ and part of the OFF-set of $\mathcal{F}$ (red bullet in Figure 2-a). As mentioned above, this is what we refer as the *quasi-exact* representation. In order to achieve an exact representation, $\mathcal{I}$ has to be flanked by an extra function $\mathcal{S} : \mathbb{B}^3 \rightarrow \mathbb{B}$ which fires *iff* function $\mathcal{I}$ infers a wrong output value. A possible expression for $\mathcal{S}$ is given by (3).

$$\mathcal{S}(a,b,c) = \neg a \vee (\neg b \wedge \neg c) \tag{3}$$

It is therefore clear that $\mathcal{I}$ and $\mathcal{S}$ should collaborate to achieve the exact representation of $\mathcal{F}$. Indeed, $\mathcal{S}$ plays the role of a *Supervisor* that can be used as a flag to decide whether $\mathcal{I}$ has to be corrected.

The composite function $\widetilde{\mathcal{F}} = \mathcal{S} \circ \mathcal{I}$ can be graphically represented as the projection of the cube of $\mathcal{I}$ (Figure 2-b) into the 4^{th} dimension $\mathcal{S}$ (Figure 2-c). The resulting hypercube is described by the following algebraic expression:

$$\widetilde{\mathcal{F}}(a,b,c,\mathcal{S}) = \mathcal{S} \wedge \neg a \vee \neg \mathcal{S} \wedge a \tag{4}$$

under the following satisfiability don't care (SDC) condition:

$$SDC(\mathcal{S}) = \neg \mathcal{S} \wedge \mathcal{F} \oplus \mathcal{I} \tag{5}$$

e.g., $\{\neg \mathcal{S} \wedge \neg a\}$ in our example. Such SDC is imposed by construction: $\mathcal{S}$ fires *iff* $\mathcal{I}$ is wrong.

$\widetilde{\mathcal{F}}$ can be graphically simplified by: (i) collapsing minterms on the versor $\widehat{\mathcal{S}}$ on the inner cube, (ii) dropping unreachable vertices (bold dotted lines in the picture). Indeed, both $\mathcal{I}$ and $\mathcal{S}$ have same support-set. The result is the 3-D cube of $\mathcal{F}$. The equivalent Boolean operation is the substitution $\mathcal{S} \rightarrow \widetilde{\mathcal{F}}$, which leads to the final relationship in (6). This transformation brings back to the original Boolean function $\mathcal{F}$.

$$\begin{aligned}\widetilde{\mathcal{F}}(a,b,c) &= ((\neg a \vee (\neg b \wedge \neg c)) \wedge \neg a) \vee (\neg(\neg a \vee (\neg b \wedge \neg c)) \wedge a) \\ &= (a \wedge b) \vee (a \wedge c) \vee \neg a \\ &= \mathcal{F}(a,b,c)\end{aligned} \tag{6}$$

To summarize: *Any n-ary Boolean function $\mathcal{F}$ can be described as:*

$$\mathcal{F} = (\mathcal{S} \wedge \neg \mathcal{I}) \vee (\neg \mathcal{S} \wedge \mathcal{I}), \tag{7}$$

where $\mathcal{I}$ is a k-ary inferential function with $k < n$, and $\mathcal{S}$ is a n-ary supervisor function that fires **iff** $\mathcal{I} \neq \mathcal{F}$.
It is important to notice that this transformation does not impose, nor require, any specific characteristics on $\mathcal{F}$.

B. Inferential Logic

The architecture of an *Inferential Logic Circuit* (ILC), Figure 3, is a straightforward implementation of the Boolean formulation described in Section III-A. It consists of two main logical blocks: (i) the *Inferential Unit* (IU), which implements the CT function $\mathcal{I}$; (ii) the *Supervisor Unit* (SU), which restores the output of $\mathcal{F}$ when $\mathcal{I}$ yields to an incorrect prediction.

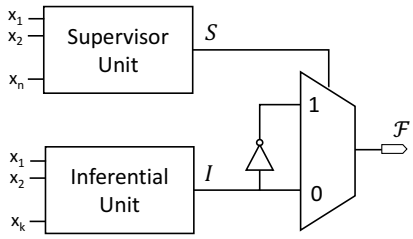

Figure 3. Inferential Logic Circuit architecture.

An ILC can run under two operating modes.
Quasi-exact computing: inputs $X_k = (x_1, \ldots, x_n)$ are evaluated by the IU; its output is *similar* to the Boolean function $\mathcal{F}$. The accuracy of the IU is defined as the ratio between the number of correctly-classified inputs and the total number of inputs, i.e., 2^n.
Exact computing: inputs $X_k = (x_1, \ldots, x_n)$ are evaluated by both the IU and the SU. Two are the possible outcomes: (i) the input pattern belongs to the set of misclassified patterns, hence the output inferred by the IU is wrong and the SU drives the complement of IU towards the primary output; (ii) the input pattern belongs to the set of correctly classified samples, hence the value inferred by IU is propagated to the output.

C. Design Flow

Figure 4 gives an overview of the proposed design flow. It starts with an exhaustive description of the digital circuit defined through a truth table (2^n permutations). Such table represents the training set. Building the CT is demanded to a Matlab script that leverages the `fitctree` function. Once the CT is generated, the validation phase takes place; each entry of the truth table is classified by means of the CT and results are collected into two separate sets: the classified set $\mathcal{C}$, and the misclassified set $\mathcal{M}$. The former is a Boolean description of the CT, that is, the function $\mathcal{I} : \mathbb{B}^k \rightarrow \mathbb{B}$. By means of an in-house software package, described later in Section IV-A, the CT structure is synthesized, optimized and mapped onto a MUX-INV netlist. The resulting logic circuit is the *Inferential Unit* of Figure 3. $\mathcal{M}$ is a Boolean description of the *Supervisor Unit* (SU), i.e., the function $\mathcal{S} : \mathbb{B}^n \rightarrow \mathbb{B}$, and it is synthesized with the ABC synthesis tool [9]. According to (7), output signals of the Inferential and Supervisor units are then plugged into a MUX. The obtained ILU architecture is

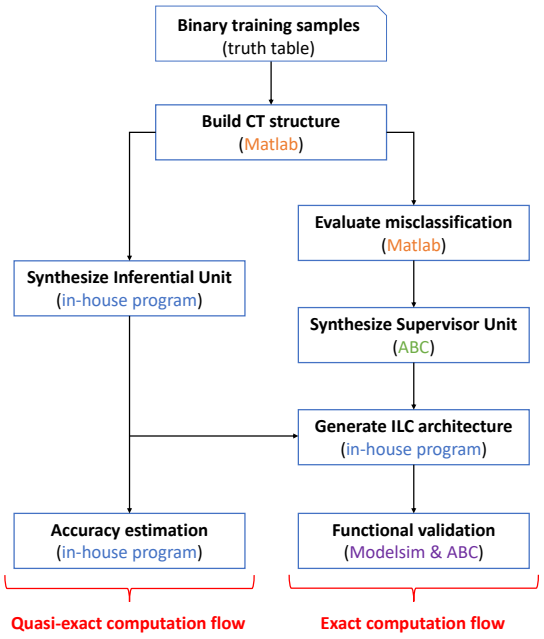

Figure 4. Design flow generating Exact and Quasi-Exact Computing Flows.

then validated thought exhaustive functional simulations and Boolean equivalence checking.

IV. Logic Synthesis of the Inferential Unit

As previously discussed, the Supervisor is synthesized with a standard multi-level flow. Here we focus our discussion on an efficient synthesis flow tailored on quasi-exact Boolean function representations.

A. From CTs to Mux-Inverter Trees

A CT is a rooted and directed acyclic graph defined as $\Gamma = (\Phi \cup D \cup \Theta \cup E)$. The set of decision vertices $d \in D$ with two outgoing edges $e_0, e_1 \in E$ are associated with a threshold operation on a primary input variable $x_k \in \mathcal{X}$. Such nodes drive the connection between nodes at the lower levels with those at the upper levels of the tree structure. For instance, if $x_k > T_h$ (being T_h the threshold selected at training time by the Gini index), then the left branch is connected to the current node's output, right branch otherwise. Terminal nodes $\theta \in \Theta$ with out-degree 0 represent the class to which a given input pattern belongs to. Finally, root nodes $\phi \in \Phi$ with in-degree 0 represent the output value assumed by the Boolean logic function $\mathcal{F}(\mathcal{X})$. From the definition above, the representation of decision nodes using Boolean gates may look prohibitive due to the complexity involved in designing a threshold comparator. However, since training samples are always described by means of Boolean variables, the CT algorithm will always select a threshold subject to (8), where $max(k)$ and $min(k)$ represent the maximum and minimum values assumed by the input feature k, namely 1 and 0.

$$T_h = \frac{max(k) + min(k)}{2} = 0.5 \tag{8}$$

It is thereby possible to reduce the threshold comparator through the equivalences reported in (9).

$$\begin{aligned}(x_k \geq 0.5) &\equiv (x_k = 1)\\ (x_k < 0.5) &\equiv (x_k = 0)\end{aligned} \tag{9}$$

Such equations suggest that decision vertices can be reduced to a simple identity comparator, not dissimilar to decision nodes in Binary Decision Diagrams (BDDs) [10]. The built-in functionality can then be reduced to an If-Then-Else (ITE) primitive, where the selection variable is represented by the primary input $x_k \in \mathcal{X}$, and the *true* and *false* branches are represented by the left and right child nodes, respectively. As a consequence, the most simple transposition of a decision node is a single Multiplexer (MUX). Without loss of generality, we can say that any CT grown on a fully-Boolean training sample can be represented by means of a tree of MUXes. Obviously, other circuit implementations do exist.

B. Synthesis Flow

Figure 5 describes the adopted synthesis flow for the inferential unit. First, the CT structure described as a list of

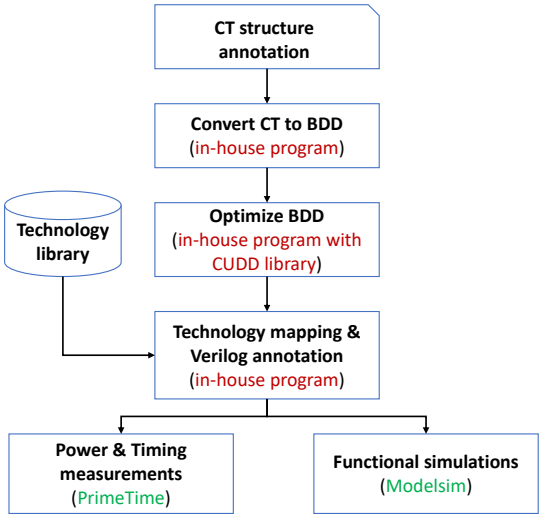

Figure 5. Synthesis flow for the Inferential Unit.

ITE statements is converted into a BDD structure and then optimized according to the following three rules:

1) Node elimination: a node having both left and right branches connected to the same child node represents a don't care. Therefore, it is eliminated from the structure.
2) Node sharing: if two, or more, nodes share the same child nodes, then they are said to be equivalent. It is then possible to merge them into a single node.
3) Variable ordering: an good variable order is selected as to guarantee the lowest cardinality for the decision diagram [11].

Those transformations are performed with an in-house C program that leverages the CUDD package [12] for decision diagram manipulations.

As a final stage, the reduced and ordered BDD is mapped into a MUX-INV tree, where MUXes are used to map decision nodes, and INVs are used to eliminate constant terminal nodes and to invert signals along negated edges. The obtained annotated Verilog netlist is then validated through functional simulations and post-synthesis analyses.

V. Experimental Results

In this section we demonstrate the effectiveness of the proposed ILC paradigm. First, we show that Inferential Computing is particularly effective in error-resilient applications; we provide a case-study analysis for a widely-used edge detection technique, i.e., the Sobel operator [13]. We describe in detail the adopted implementation and provide an accurate analysis of the obtained results in terms of output accuracy, area, and power efficiency. Second, we quantify area and power of ILC circuits against an open-source multi-level synthesis flow: the ABC synthesis tool [9]. Quality-of-result for quasi-exact and exact computing is also assessed.

A. Edge Detection Through The Sobel Operator

In order to appreciate the potential of the proposed technique, we resort to a real-life error-resilient application. In the field of machine vision, edge detection is the operation of detecting significant local changes in an image. Given a matrix of pixels I, a *step edge* is associated to a peak in the first derivative of I. Edges can be detected by means of a gradient operator. In our analysis, we employee the Sobel operator, defined as in (10).

$$G = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \tag{10}$$

Let us assume to have a 3x3 gray scale image I, where each element $i_{x,y}$ of the matrix represents a pixel in the image. The Sobel operator G applied to I returns the matrix convolution: an element-wise multiply and accumulate function, as in (11); the result of such convolution represents the intensity of the pixel stored in the output matrix C at position (x,y).

$$\begin{aligned} C_{x,y} = I \otimes G &= \begin{pmatrix} i_{x-1,y-1} & i_{x-1,y} & i_{x-1,y+1} \\ i_{x,y-1} & i_{x,y} & i_{x,y+1} \\ i_{x+1,y-1} & i_{x+1,y} & i_{x+1,y+1} \end{pmatrix} \otimes G \\ &= i_{x-1,y-1} \cdot 1 + i_{x-1,y} \cdot 0 + \cdots + i_{x+1,y+1} \cdot (-1) \end{aligned} \tag{11}$$

Such operation is then iterated over all the pixels contained in I, thus to generate a new image containing information on horizontal edges.

The circuit we implemented performs the convolution reported in (11) between G and a gray scale source image where each pixel is in the range $[0-255]$. The input of the circuit is composed of the value of the 6 pixels involved in the convolution. Input space cardinality was reduced by considering only the three most significant bits for each pixels, with a total of 18 primary inputs. A threshold operator on the result of (11) was also applied as to eliminate noise on the resulting images. As a consequence, the output of the circuit is represented by a single bit subject to the following rule: if C is below the intensity threshold, then the output pixel does not represent an edge (output 0); otherwise the function evaluates to 1.

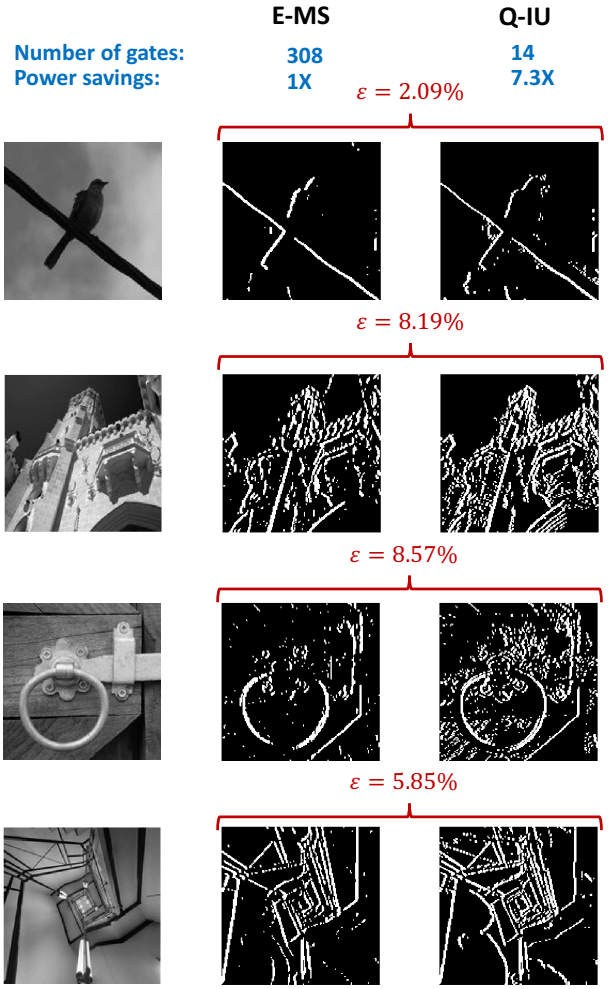

Figure 6. Result comparison with circuit implementing the Sobel operator. Original image (left), E-MS (center), Q-IU (right).

The exhaustive truth table was then generated by means of a Matlab script. The experimental setup is composed of two design flows, summarized as follows.

Quasi-Exact Computing via Inferential Unit (Q-IU): only the Inferential Unit composing the architecture illustrated in Section III is employed. Mapping and optimization as described in Section IV are also applied.

Exact Computing via Multi-level Synthesis (E-MS): the benchmark is processed with the ABC synthesis tool and mapped on a technology library at the 45nm node consisting of 32 logic primitives. The design is optimized by using the built-in `collapse` command. We first analyze the quality-of-result of the Q-IU implementation. Figure 6 shows a few sample images[1] of size 150×150 pixels used for the evaluation process. Error rate (ε) was computed as the pixel-wise difference between the exact solution provided by the E-MS circuit, and the quasi-exact counterpart obtained through the Q-IU. Starting from the topmost picture in Figure 6, error rates for each image are: 2.09%, 8.19%, 8.57%, and 5.85%.

[1]Considered images were obtained through an open-source repository. All images are released as public domain.

These values lead to a fundamental observation: Inferential logic is actually capable to extrapolate the most significant operations of a Boolean function, i.e., the ones with a higher computational and expressive power. This observation is supported by comparing the performance of the two implementations: the Q-IU is composed of 14 gates only, thus ensuring a 22× smaller area w.r.t. the E-MS counterpart. Such a huge area saving translates to 7.3× less dynamic power consumption.

B. ILC vs. Multi-level

It is important to understand that the objective of this section is not to demonstrate how ILC circuits could replace standard synthesis flows, but rather to provide a clearer picture of pro's and con's of adopting the inferential paradigm in accuracy-critical applications. The adopted experimental setup can be summarized as follows.

ILC performing exact computing (E-ILC): benchmarks are represented with the architecture illustrated in Section III; the Inference Unit is designed by means of the mapping and optimization steps described in Section IV; the Supervisor Unit is obtained with a standard multi-level ABC synthesis flow. Notice that the design of the Supervisor Unit is not optimized, as to provide a worst-case analysis. Multi-output logic functions are elaborated by isolating and processing each output cones separately.

ILC performing quasi-exact computing (Q-ILC): in this case each benchmark is composed of the Inference Unit alone; the design encompasses the stages described in Section IV.

Multi-level Synthesis (MS): benchmarks are processed with the ABC synthesis tool. Each benchmark is optimized by means of the built-in `fx` command.

Considered benchmarks are general-purpose open-source circuits belonging to the LGSynth91 suite. Designs are elaborated by means of their exhaustive truth table descriptions with digital files compliant to the Espresso format definition. In all considered synthesis flows we adopt the same CMOS technology library at the 45 nm node consisting of 32 logic primitives. Power estimations are conducted by means of Synopsys PrimeTime simulations. It is important to underline that truth tables represent a readily available circuit description for our purposes; obviously, they represent a bottleneck, especially with big circuits. However, the objective of this work is not to provide an ultimate solution, but rather to illustrate a proof-of-concept for future ML-driven design flows. Alternative design methodologies to overcome this issue, e.g., derive CTs from RTL descriptions, are part of future works.

Figure 7 shows the accuracy of both E-ILC and Q-ILC implementations; numbers have been obtained by means of exhaustive functional simulations for each considered benchmark. The plot demonstrates the E-ILC architecture yields designs which are fully compliant with the original Boolean network specifications. On the other hand, Q-ILC designs achieve remarkable accuracy: 90% on average.

Table I reports the total number of devices for each benchmark processed with the three considered implementations. Notice that the area of the E-ILC is due to the contributions of both the Supervisor and the Inferential units. Numbers suggest that

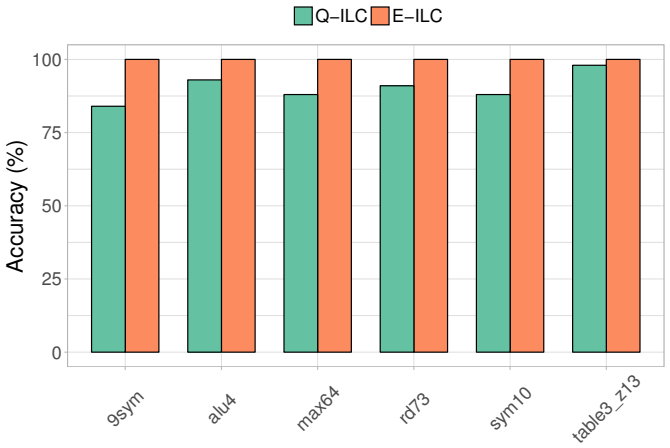

Figure 7. Quality of result analysis: E-ILC vs. Q-ILC.

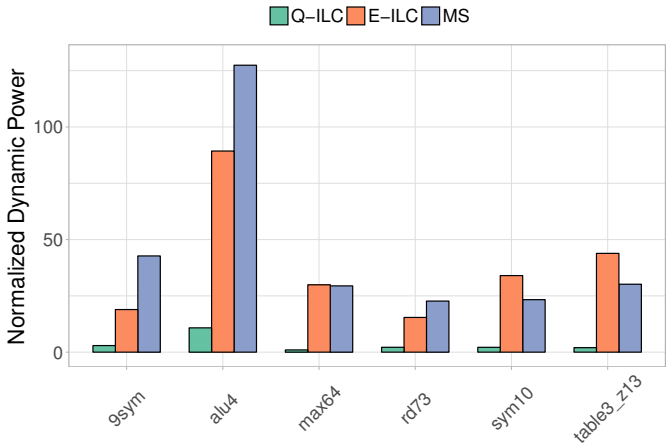

Figure 8. Normalized dynamic power for the three considered design flows.

Table I
OBTAINED NUMBER OF DEVICES WITH Q-ILC, E-ILC, AND MS DESIGN FLOWS. REPORTED SAVINGS ARE W.R.T. MS.

	ILC					MS
	Q-ILC	Savings (%)	Supervisor	E-ILC	Savings (%)	
9sym	13	87.61	52	65	38.09	105
max64	8	91.11	100	108	-20	90
rd73	18	73.13	42	60	10.44	67
sym10	10	93.71	107	117	26.41	159
table3_z13	9	93.83	137	146	0	146
alu4	288	84.72	1409	1697	9.97	1885
Average	**57.66**	**87.35**	**307.83**	**365.5**	**10.56**	**408.66**

E-ILC implementations achieve excellent results. Indeed, E-ILC shows an average gate count that similar to the MS counterparts (10.56% smaller). As one might observe, there are cases where area gets larger, e.g., the `max64`. Nonetheless, these results suggest that the proposed ICL architecture can allow to achieve acceptable results in terms of area occupation. On the other hand, the portion of area taken by the Q-ILC is substantially smaller, with an average gate saving of 87.35%. Another important concern relates to power consumption. Figure 8 depicts the normalized dynamic power recorded for each implementation. On average, the Q-ILC implementation achieves 13× (10×) lower dynamic power consumption w.r.t. MS (E-ILC). The E-ILC almost matches the power profile of the MS implementation, still with a 16% saving on average. Also in this case, there might be benchmarks for which E-ILC consumes more power, e.g., `max64`.

It is worth to emphasize that, even without aggressive logic optimizations, ILC circuits show similar performance of multi-level counterparts, yet with an additional feature: the possibility to readily adopt an on-line, adaptive, trade-off strategy between accuracy and power consumption. Indeed, through a dedicated *power-knob*, ILCs could switch between (i) quasi-exact computations, for devices with scarce power budgets, e.g., a mobile device running out of power, as to achieve minimal power consumptions still having a good quality of result, or (ii) exact computations, which can be used whenever accuracy scaling is not an option.

VI. CONCLUSIONS

In this paper we introduced for the first time a novel design paradigm for combinational circuits, the *Inferential Logic Circuits*. The proposed ML-inspired synthesis flow yields digital circuits that leverage statistical inference to process the outcome. Experimental results demonstrate that the proposed quasi-exact computation is a viable solution for error-resilient applications; it achieves 90% accuracy using 87.35% fewer devices w.r.t. a multi-level logic design; the same circuits can also deliver exact computations with 100% accuracy.

REFERENCES

[1] H. Brink, J. Richards, and M. Fetherolf, *Real-world machine learning*. Manning Publications Co., 2016.

[2] L.-C. Wang and M. S. Abadir, "Data mining in eda - basic principles, promises, and constraints," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. ACM, 2014, pp. 159:1–159:6.

[3] L.-C. Wang, "Experience of data analytics in eda and testprinciples, promises, and challenges," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 6, pp. 885–898, 2017.

[4] O. Guzey *et al.*, "Extracting a simplified view of design functionality based on vector simulation," in *Haifa Verification Conference*. Springer, 2006, pp. 34–49.

[5] V. Tenace and A. Calimera, "Activation-kernel extraction through machine learning," in *CAS (NGCAS), 2017 New Generation of.* IEEE, 2017, pp. 5–8.

[6] R. G. Rizzo, V. Tenace, and A. Calimera, "Multiplication by inference using classification trees: A case-study analysis," in *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*. IEEE, 2018, pp. 1–5.

[7] A. J. Izenman, "Modern multivariate statistical techniques," *Regression, classification and manifold learning*, 2008.

[8] J. L. Gastwirth, "The estimation of the lorenz curve and gini index," *The review of economics and statistics*, pp. 306–316, 1972.

[9] B. L. Synthesis and V. Group, "ABC: A System for Sequential Synthesis and Verification," http://www.eecs.berkeley.edu/ alanmi/abc/, 2016.

[10] S. B. Akers, "Binary decision diagrams," *IEEE Trans. Computers*, vol. 27, no. 6, pp. 509–516, 1978.

[11] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys (CSUR)*, vol. 24, no. 3, pp. 293–318, 1992.

[12] F. Somenzi, "CUDD: CU decision diagram package-release 2.4. 0," *University of Colorado at Boulder*, 2009.

[13] R. Jain, R. Kasturi, and B. G. Schunck, *Machine vision*. McGraw-Hill New York, 1995, vol. 5.