Received November 4, 2019, accepted November 29, 2019, date of publication December 10, 2019,
date of current version December 23, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2958605

Formal Methods for Exact Analysis of Approximate Circuits

ZDENEK VASICEK, (Member, IEEE)
IT4Innovations Centre of Excellence, Brno University of Technology, 60190 Brno, Czech Republic
(e-mail: vasicek@fit.vutbr.cz)

This work was supported by the Czech Science Foundation under Project 19-10137S.

ABSTRACT Approximate circuits are digital circuits that are intentionally designed in such a way that the specification is violated in terms of functionality in order to obtain some improvements in power consumption, performance or area, in comparison with fully functional circuits. To design the approximate circuits, the synthesis tools rely on the availability of a procedure checking, whether the synthesized circuits meet a specification and/or provides information about circuit quality. Compared to the traditional circuit design flow, the nature of the approximate circuits involves replacing the strict functional equivalence checking with a more advanced approach that enables us to quantify or guarantee the degree of similarity. The most common technique is to employ a circuit simulator for analysing responses for all input vectors. This approach allows us to simultaneously perform checking and quality assessment, but the exhaustive enumeration of the input vectors is tractable only for a small number of inputs. To avoid excessive run-times, a subset of all possible input vectors is typically used for complex circuits. This causes us, however, to lose the ability to guarantee that the quality of the synthesized circuits is within an acceptable range given in the specification. The main goal of this paper is to show how to adopt formal methods such as binary decision diagrams and satisfiability solvers for exhaustive analysis of approximate circuits without explicit enumeration of all input vectors. We survey the methods for exact computation of the most important error parameters used in the context of approximate computing, propose improved algorithms and provide a detailed analysis of their performance. The methods are benchmarked on a large set of key approximate circuits consisting of nearly 2,000 unique arithmetic instances with 8-, 12-, 16-, and 32-bit operands which helps us to identify the best algorithm and method for computation of a desired error parameter.

INDEX TERMS Approximate computing, approximate circuits, binary decision diagrams, error metrics, error analysis, formal methods, satisfiability solvers, quality assessment.

I. INTRODUCTION

Recently, approximate computing has been introduced as a technique for addressing the relentless rise in demand for energy-efficient systems. The concept of approximate computing exploits the idea of accepting a certain level of inaccuracy in computations in order to reduce complexity and improve other parameters of digital systems (such as power consumption, speed, etc.) The key motivation behind approximate computing is the inherent error resilience of many real-world applications. The inherent error resilience in fact means that it is not always necessary to implement precise and usually area-expensive circuits. Instead, much simpler, approximate circuits may be used in a given application without introducing any significant degradation in results produced by this application.

The associate editor coordinating the review of this manuscript and approving it for publication was Dušan Grujić.

The approximate computing has been applied on the whole digital system stack and many different approaches have been introduced in the last decade from hardware- to application-level. A good survey of existing techniques can be found, for example, in [1]–[3]. Technology-independent functional approximation currently represents the most popular technique on how to introduce approximations to hardware components. Apart from that, voltage over-scaling or overclocking can be applied to introduce errors. The idea of functional approximation is to implement a slightly different function to the original one, provided that the accuracy is kept at a desired level and the power consumption or other electrical parameters are optimized adequately. The functional approximation can be performed manually, but the current trend is to develop fully automated functional approximation methods that can be integrated into computer-aided design tools for digital circuits. The goal is to obtain a tool performing an automatic approximation of digital circuits independent of their structure. A list of the available methods can be found, for example, in [3].

The problem is that the automated synthesis of approximate circuits is, in fact, a multi-objective optimization problem in which a circuit satisfying user-defined constraints and showing the desired trade-off between the quality and other electrical parameters is sought in the space of all possible implementations. The circuit approximation is, however, in principle an incompletely specified problem. It means that the target logic function implementing the required behavior is unknown a priori. As we cannot specify the problem, we cannot easily employ standard synthesis tools to design such circuits. Hence, the approximate methods are typically constructed as iterative methods performing a design space exploration. In each iteration, several candidate approximate circuits are generated and evaluated in terms of functional (quality) and non-functional requirements (electrical parameters) [4]–[6]. While there are common approaches on how to estimate and evaluate the electrical parameters (area, delay, and power consumption), determining the quality of a candidate approximation is, in general, a nontrivial problem, especially when we consider the fact that it must be done not only as quickly as possible because of its integration into an iterative design process, but also as accurately as possible in order to obtain an optimal, or at least a close to optimal, approximation.

The quality of approximate circuits can be calculated by means of a circuit simulator. The circuit simulator is employed to determine responses for all input vectors that are then used for error computation. Unfortunately, the exhaustive enumeration of the input vectors is tractable only for a small number of inputs. Researchers typically cope with the limited scalability by using a subset of all possible input vectors. This causes us, however, to lose the ability to guarantee that the quality of the synthesized circuits is within an acceptable range given in the specification. Using a subset of all possible input vectors can cause a substantial difference between the estimated and real error [7], [8]. Hence, various analytical and formal approaches have been proposed and applied for exact quantification of the error of the approximate circuits [9]–[19]. In this paper, we will focus on the formal approaches because they do not make any assumption on the structure of the approximate circuits and they can be applied to determine almost every error metric.

Unfortunately, the formal algorithms developed for exact error analysis of approximate circuits are typically presented without a detailed evaluation. It is thus not clear which algorithm performs better and which one is worse. In addition, the authors typically evaluate neither the scalability nor robustness. The situation gets complicated especially when completely different approaches are used (e.g. methods based on binary decision diagrams vs. those utilizing satisfiability solvers). There is no clue on how to choose the most convenient approach.

A. GOALS AND CONTRIBUTIONS OF THIS WORK

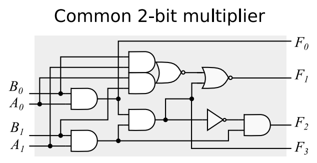
The goal of this work is to summarize the current state-of-the-art related to the formal error analysis of approximate circuits and conduct a comprehensive analysis of the proposed error analysis methods on a set of benchmark circuits. The goal is to evaluate the performance, robustness, and scalability of each method. In particular, we focus on the mainstream formal methods based on satisfiability (SAT) solvers and Binary Decision Diagrams (BDDs). In addition, the simulation-based method is evaluated.

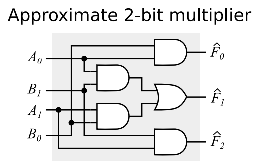
The work in this paper extends the basic notions given in [15] and makes the following key contributions:

1) We provide an extensive performance analysis of several approaches for determining the error of approximate circuits. An exhaustive simulation, a formal analysis based on BDDs and a formal analysis based on satisfiability solvers is investigated. The methods are benchmarked on a set consisting of nearly 2,000 different approximate arithmetic circuits.
2) We present a complete overview of the error metrics proposed in the literature in order to evaluate the quality of approximate circuits. For each metric, a corresponding computation algorithm is derived. The algorithms are formulated at an abstract level which helps us to understand the theoretical computational complexity and receive a universal recipe on how to determine the error regardless of the used formal apparatus.
3) We present novel algorithms for determining the worst-case Hamming distance, the mean squared error and the worst-case relative error. In addition, an improved algorithm for computing the absolute worst-case error is proposed.
4) For each error metric, the best algorithm and formal approach is identified. In addition, we discover the importance of the worst-case absolute error metric and its role as an indicator of the complexity of error analysis.

B. PAPER ORGANIZATION

Section II introduces the problem of determining the error of approximate circuits and surveys existing work related to this research problem. Properties of various methods for exact as well as approximate analysis of approximate circuits are discussed in this part. Section III provides an overview of various error measures typically employed in the literature and contains a formal definition of the corresponding error metrics. In addition, notation used within this paper is summarized. Section IV firstly presents an overview of the methods for the exact error analysis and discusses their computational complexity. Then, the algorithms for computation of the error metrics are derived. The performance of the algorithms is





F (B_1B_0 \ A_1A_0)	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	1001

$\hat{F}$ (B_1B_0 \ A_1A_0)	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	0111

FIGURE 1. Example of 2-bit approximate multiplier $\hat{F}$ a) Truth table and b) corresponding gate-level implementation.

experimentally evaluated in Section V. Finally, conclusions are given in Section VI.

II. RELATED WORK

A. EVALUATING FUNCTIONALITY OF APPROXIMATE CIRCUITS

Arithmetic circuits are a natural target for approximation methods. Our intuition suggests avoiding introducing any approximations in the more significant bits; however, this does not have a rational reason. Figure 1 shows a 2-bit approximate multiplier where the last row of the Truth table (i.e. the case when both inputs are equal to three) is modified in such a way that the correct product nine was replaced with seven [20]. This modification enables us to reduce some logic and remove the most significant bit associated with the weight 2^4. Despite that, the error magnitude is relative low; it equals $9 - 7 = 2$. This example demonstrates that we cannot simply look at the isolated bits and their significance (weight), and the whole context should be investigated instead. This makes the error analysis non-trivial and computationally complex.

The function of approximate circuits is typically expressed using one or several error metrics. In addition to the error magnitude, the average-case, as well as the worst-case situation, should be analyzed. Among others, mean absolute error and mean square error are the most familiar metrics related to the average-case analysis. Selection of the right metrics is a key step of the whole design. When an arithmetic circuit is approximated, it is necessary to base the error quantification on an arithmetic error metric since the error magnitude could have a significant impact on the target application. For general logic, where no additional knowledge is available and where there is not a well-accepted error model, Hamming distance or error rate is typically employed. In signal processing applications, peak signal-to-noise ratio (PSNR) is typically used to evaluate the impact of a given algorithm on the signal quality. As the PSNR is defined via the mean squared error, it suggests using mean squared error as a quality indicator. The complexity of determining the most suitable metrics, however, was demonstrated by Miao et al. who presented a formal proof showing that for signal processing applications the quality-optimal approximate addition is achieved by limiting maximum error magnitude while accepting a larger error rate [21].

In some cases, neither the Hamming distance nor the arithmetic metrics provide a satisfactory assessment of the quality of approximate circuits. Hence, various problem-specific error metrics have been introduced. For example, a data-independent metric denoted as distance error was proposed to model the error introduced by the approximations of the median and sorting networks [22], [23]. The authors proposed measuring the distance between the rank of the returned element and the rank expected by the specification. Other metrics were proposed to quantify the functionality of approximate circuits when used in the sequential circuit [24]. In particular accumulated worst-case error and accumulated error rate have been introduced. These metrics are motivated by the fact that the worst-case computed for the approximate component in isolation may differ substantially from the accumulated worst-case when the component is used in a sequential circuit.

In addition to the functionality, we can investigate parameters related to the testability of the approximate circuits. During the manufacturing process, various physical defects can affect the integrated circuits and may be the cause of faults leading to observable errors. The errors due to faults may affect outputs, but these errors may be still tolerable because the defect circuit does not violate the specification. From this point of view, we can investigate the complexity of the testing in terms of the number of input combinations required to test a given approximate circuit [25]. Thanks to fewer required test vectors, we can achieve test-cost reduction and improvements in yield.

B. EXACT ERROR ANALYSIS OF APPROXIMATE CIRCUITS

The functionality of approximate circuits can in principle be quantified by comparing the truth table of the approximate and accurate implementations. The most straightforward method for enumerating the truth table of an approximate implementation is to use a circuit simulator. The simulator-based approach is flexible as it can be used to compute an arbitrary error metric independently of the structure or principle of the analyzed approximate circuit. The exact error can be obtained by means of an exhaustive simulation analyzing responses for all possible input vectors. The exhaustive simulation is, however, time-consuming because the number of input vectors grows exponentially with the number of inputs. As we will show in Section V, this approach is thus applicable for circuits having up to 32 inputs (e.g. arithmetic circuits with two 16-bit operands). It is very important to choose a suitable simulator to maximize the performance. Otherwise, the simulation can become impractical even for 8-bit circuits. This can easily happen when a Matlab circuit model is used instead of a gate-level description and a less generic, but more efficient gate-level simulator. Twenty-five hours are reported when analyzing 14-bit adders in [26] and more than 19 years are estimated when performing the exhaustive simulation of 20-bit adders [9].

So as to avoid excessive run-times and enable the analysis of complex circuits, many authors simplify the problem and evaluate the functionality of approximate circuits by applying a subset of all possible input vectors. They perform, for example, Monte Carlo simulation to measure the error of the output vectors with respect to the exact solution [5], [10], [27]. Similar to the exhaustive simulation, the approach can be time-consuming depending on the chosen number of vectors and a particular simulator. To provide an example, six seconds were required to perform the Monte-Carlo simulation of various 32-bit adders and 8-bit multipliers on a machine with 2.66 GHz CPU [10]. When a subset of all possible input vectors is evaluated, the error is only estimated. The actual error value can be underestimated, as well as overestimated, depending on the chosen vectors. The lack of any formal guarantee then decreases the credibility of the approximate circuits. It has been mathematically derived that the difference between the values obtained using the Monte-Carlo approach and the exact error may be substantial. For the truncated adders characterized by the maximum carry chain length k, for example, the difference increases exponentially with k and is equal to 12% for $k = 10$ [7]. For the 32-bit block-based approximate adders, the Monte-Carlo method with the sample size equal to 10K and 100K exhibits the relative error larger than 33% and 11.2%, respectively [8].

In order to address the inaccuracy of the Monte-Carlo simulation, various statistical and analytical methods have been proposed [7]–[9], [11], [12]. The common feature of these techniques is that the error is expressed as a function of circuit parameters like the number of inputs, the number of truncated bits, carry-chain length, the number of subcomponents, etc. The inputs are treated as random variables and various theorems of probability theory are applied to exactly describe, or at least estimate, the error statistics and the probability mass function of the error [9]. The accurate mathematical formulation of the error is possible only for some error metrics and those approximate circuits that typically exhibit a regular structure. Hence, some analytical methods are limited to a particular variant of approximate circuits such as truncated multipliers or broken-array multipliers [12]. Others are applicable to a broader range of circuits such as approximate adders consisting of sub-adder units [8], [26] or multipliers recursively constructed from smaller multipliers [9], [12]. The analytical models are fast but the complexity of their derivation increases with the increasing model granularity due to the presence of non-trivial conditional probabilities. This means that it is currently impossible to construct an accurate yet simple mathematical model for circuits described at the level of gates.

The limitations of the previously mentioned methods can be overcome by using formal methods such as satisfiability (SAT) solvers, Binary Decision Diagrams (BDDs) and other techniques used to verify hardware systems. The formal approaches do not make any assumptions regarding the structure of the approximate circuits. They can be applied to determine almost every error metric and they always produce the exact error value. Except for some pathological cases of circuits, the formal techniques are fast and allow for the analysis of circuits having hundreds of inputs. Instead of the common functional equivalence checking, however, relaxed functional equivalence checking is requested stressing the fact that the approximate circuits match the specification up to some bounds with respect to a chosen error metric. As the exact error analysis becomes more and more important, many different techniques have been recently proposed to address the problem of relaxed equivalence checking in the approximate computing. The first technique for formal analysis of approximate circuits was based on a SAT solver combined with an ILP solver [10]. Due to a limited applicability (an algorithm for analysis of the worst error was proposed only) and scalability coming from the usage of ILP, the researchers then focused mainly on the usage of BDDs. Several BDD-based methods were introduced to calculate a broader spectrum of error metrics [10], [13]–[15]. At present we have a variety of more efficient algorithms based on SAT solvers [16], [17]. Besides these main-stream approaches, some other techniques have been developed. For example, methods based on model checking were introduced for an error analysis of sequential circuits that contain approximated combinational components [24]. Due to scalability issues, the results were presented for small sequential multipliers based on an 8-bit adder and some less complex circuits only. In addition, Symbolic Computer Algebra (SCA) has been used to determine various error metrics [18], [19]. The latter approach uses Algebraic Decision Diagrams (ADDs) that has the ability to determine the mean relative error for which no other formal technique exists. The usage of SCA, however, suffers from limited scalability. The results were presented for 8-bit and 16-bit adders only. The analysis of multipliers seems to be intractable even for 8-bit instances. The problem of the SCA on harder circuit instances is that the polynomial representations may lead to an enormous remainder describing the difference between an approximate and reference circuit whose further analysis is intractable.

The knowledge of exact error is important not only for time-critical and dependable systems, but also for not so critical applications such as image processing, where a low average error but excessive worst-case error can produce unacceptable results [28]. Obviously, the lack of formal proof limits the application of approximate computing in practice. However, there are also other reasons that increase the demand for fast yet exact error assessment. If an error analysis is employed within the optimization algorithm (i.e. the error is used to guide the search through the search space as in [13], [16]), the analysis method must be simple in order to compute it because it is called very frequently. In addition, the computed value must be credible to allow for discovery of energy-optimal solutions and avoid premature convergence due to discrepancies between calculated and real error [6]. Finally, the high-level-synthesis methods typically rely on the availability of a library of pre-characterized basic building blocks [11], [29]. The accuracy of these methods thus depends on the quality of the error parameters available in the library.

Our work focuses on the usage of the formal techniques for error computations, but the proposed algorithms can also be adopted for automatic test pattern generation (ATPG). In fact, the problem formulations of SAT and ATPG are closely related [30]. It is the task of an ATPG-algorithm to generate a test for every fault in the circuit according to some fault model. The ATPG problem have been investigated intensively for many decades and many powerful approaches have been proposed. However, the conventional ATPG algorithms, when not aware of the approximation, do not distinguish whether a particular fault is tolerable or not. This can lead to a rejection of fabricated circuits that can still be considered as non-faulty because of approximation. Several works dealing with the approximation-aware ATPG exist in the literature. In [31], for example, a SAT solver is used to identify and remove all faults that no longer need to be tested because they can be tolerated under the given error metric. While [31] considers worst-case arithmetic error and bit-flip error, the authors of [25] target mean absolute error and mean squared error. Looking at the structure of the architecture for filtering of the redundant faults proposed in the latter work, we can identify a common approximate miter circuit used to quantify the error (see Figure 4 and 5).

III. PRELIMINARIES

The following paragraphs introduce the notation that will be used for the remainder of this paper. They also summarize the most frequent error metrics employed in the literature and provide an overview of three Boolean problems relevant to this paper. For each metric, a formal definition is provided. These definitions are later used to derive practical algorithms for error analysis.

A. NOTATIONS AND BASIC DEFINITIONS

Let $f : \mathbb{B}^n \to \mathbb{B}^m$ be an n-input m-output Boolean function that describes correct functionality (specification) and $\hat{f} : \mathbb{B}^n \to \mathbb{B}^m$ be an approximation of it, both implemented by two circuits, namely F and $\hat{\text{F}}$, where $\mathbb{B} = \{0, 1\}$. For simplicity and without loss of generality, we assume that both F and $\hat{\text{F}}$ have the same number of outputs. Let f_i denote the i-th (counted from zero) output of an m-output Boolean function f.

Definition 1: Let $[\![P]\!]$ denote the Iverson bracket defined as $[\![P]\!] = 1$ iff the proposition P is satisfied and $[\![P]\!] = 0$ otherwise.

Considering definition 1, it holds that

$$[\![f(x) \neq \hat{f}(x)]\!] = \bigvee_{i=0}^{m-1} f_i(x) \oplus \hat{f}_i(x), \qquad (1)$$

TABLE 1. The most typical measures and the corresponding error metrics for analysis of approximate circuits.

mea-sure	error metric	
	worst case	average case
ED	worst-case absolute error (e_{wce})	mean absolute error (e_{mae})
ED^2	worst-case squared error (e^2_{wce})	mean squared error (e_{mse})
RED	worst-case relative error (e_{wcre})	mean relative error (e_{mre})
HD	bit-flip error (e_{bf})	mean Hamming dist. (e_{mhd})
EQ	common equivalence checking	error probability (e_{prob})

where $\vee$ and $\oplus$ denote common logic OR and XOR operation, respectively.

Definition 2: Let nat(x) represent a function nat : $\mathbb{B}^m \to \mathbb{Z}$ returning a decimal value of the m-bit binary vector x.

Typically, a natural binary representation is considered, i.e. $\text{nat}(x) = \sum_{i=0}^{m-1} 2^i x_i$.

Definition 3: Let $x, y \in \mathbb{B}^n$ be two Boolean vectors. We say that x is lexicographically smaller than y, denoted $x < y$, if there exists an k, $0 \leq k < n$, such that $y_k > x_k$ and $x_i = y_i$ for all $i < k$.

Considering definition 2, it holds that $nat(x) < nat(y)$.

B. ERROR MEASURES

Different error measures can be adopted to quantify the difference between the outputs produced by a functionally correct design F and an approximate design $\hat{\text{F}}$. The measures considered in this paper are summarized in Table 1. In particular, Boolean equivalence (EQ), Hamming distance (HD), arithmetic error distance (ED), squared error distance (ED^2) and relative error distance (RED) are included. Each measure can be used to define error metrics for the worst-case and average-case scenario. While the Hamming distance measures the number of different bits regardless of their significance, the arithmetic error considers the significance of each bit. The metrics are independent each other. A high error rate, for example, does not imply a high worst-case absolute error and vice versa.

C. GENERAL-PURPOSE ERROR METRICS

General-purpose error metrics that are not related to the magnitude of the output of a correct or approximate circuit are applicable to any approximate circuit independently, and whether the analyzed target is a logic or arithmetic circuit.

Definition 4: Error rate, sometimes referred to as *error probability*, is defined as the fraction of inputs vectors for which the output value differs from the original one:

$$e_{prob}(f, \hat{f}) = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} [\![f(x) \neq \hat{f}(x)]\!]. \qquad (2)$$

Definition 5: The *worst-case Hamming distance* denoted also as *bit-flip error* [32] is defined as

$$e_{bf}(f, \hat{f}) = \max_{\forall x \in \mathbb{B}^n} \left(\sum_{i=0}^{m-1} f_i(x) \oplus \hat{f}_i(x) \right) \qquad (3)$$

and gives the maximum number of output bits that simultaneously provide a wrong output value.

Definition 6: The average number of erroneous output bits denoted as *average Hamming distance* can be expressed as follows:

$$e_{mhd}(f,\hat{f}) = \frac{1}{2^n} \sum_{\forall x \in B^n} \sum_{i=0}^{m-1} f_i(x) \oplus \hat{f}_i(x) \tag{4}$$

Since $e_{prob}(f_i,\hat{f}_i) = e_{mhd}(f_i,\hat{f}_i)$, Eq. 4 can be rewritten to the equivalent alternative form:

$$e_{mhd}(f,\hat{f}) = \sum_{i=0}^{m-1} e_{prob}(f_i,\hat{f}_i). \tag{5}$$

D. ARITHMETIC ERROR METRICS

The previous error metrics assume that all the bits have the same significance; however, this is not desirable in the case of arithmetic circuits such as adders or multipliers where each bit typically has a different weight.

Definition 7: The *worst-case arithmetic error* sometimes denoted as *error magnitude* or *error significance* [33], is defined as

$$e_{wce}(f,\hat{f}) = \max_{\forall x \in \mathbb{B}^n} |\operatorname{nat}(f(x)) - \operatorname{nat}(\hat{f}(x))| \tag{6}$$

Definition 8: The *average-case arithmetic error* (*mean absolute error*) is defined as the sum of absolute differences in magnitude between the original and approximate circuits, averaged over all inputs:

$$e_{mae}(f,\hat{f}) = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} |\operatorname{nat}(f(x)) - \operatorname{nat}(\hat{f}(x))| \tag{7}$$

Definition 9: The *mean squared error* is defined as the sum of squared differences in magnitude between the original and approximate circuits, averaged over all inputs:

$$e_{mse}(f,\hat{f}) = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} \left(\operatorname{nat}(f(x)) - \operatorname{nat}(\hat{f}(x))\right)^2 \tag{8}$$

Definition 10: The *relative worst-case error* uses relative error magnitude rather than the absolute error magnitude and is defined as

$$e_{wcre}(f,\hat{f}) = \max_{\forall x \in \mathbb{B}^n} \frac{|\operatorname{nat}(f(x)) - \operatorname{nat}(\hat{f}(x))|}{\operatorname{nat}(f(x))}. \tag{9}$$

Note that special care must be devoted to the cases for which the output value of the original circuit is equal to zero (i.e. the cases when the denominator approaches zero). This issue can be addressed by either omitting test cases when $nat(f(x)) = 0$, or biasing the denominator as employed in [34].

Definition 11: When we replace the error distance in the sum of Eq. 7 with the relative error distance, we can define the *mean relative error*:

$$e_{mre}(f,\hat{f}) = \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} \frac{\left|\operatorname{nat}(f(x)) - \operatorname{nat}(\hat{f}(x))\right|}{\operatorname{nat}(f(x))}. \tag{10}$$

E. NORMALIZED ERROR METRICS AND ERROR DISTRIBUTION

As the maximum arithmetic error increases with increasing the bit-width, it is not possible to directly compare value of e_{wce}, e_{mae} or e_{mse} across circuits having different bit-widths. To overcome this limitation, normalized values are typically used for this purpose. There exists more possibilities on how to normalize the values. In [35], for example, the values of e_{mae} are normalized to be within the range [0, 1). Considering this definition, the normalized metrics can be defined as follows:

$$e_{wce\%} = \frac{1}{2^m} e_{wce}, \quad e_{mae\%} = \frac{1}{2^m} e_{mae}, \quad e_{mse\%} = \frac{1}{2^{2m}} e_{mse}, \tag{11}$$

where 2^m corresponds with the range of the values at the m-bit output.

In addition to the average-case and the worst-case error value, it is also beneficial to have knowledge of the error distribution that would provide information about the probability of occurrence of errors of different magnitudes. To compute the error distribution, we need a procedure which is able to determine the number of cases for which the error introduced by approximation is greater than a given threshold. By repeating this procedure for different thresholds, we obtain the cumulative distribution function of the error [10].

Definition 12: The number of cases causing that the absolute error is greater than a given threshold $\mathcal{T}$ is defined as

$$\#_{wcegt}(f,\hat{f},\mathcal{T}) = \sum_{\forall x \in \mathbb{B}^n} [\![|\operatorname{nat}(f(x)) - \operatorname{nat}(\hat{f}(x))| > \mathcal{T}]\!] \tag{12}$$

F. BOOLEAN SATISFIABILITY AND ITS RELATION TO THE ERROR ANALYSIS

Considering all possible input combinations and corresponding outputs does not necessarily mean that we have to explicitly enumerate them. The worst-case error analysis is, in fact, a decision problem which can be formulated and solved as the Boolean satisfiability problem. For example, the worst-case error can be determined by finding the lexicographically largest solution for the difference between the output of the approximate and reference circuit. Analogically, the average-case error analysis can be translated to the model counting problem because it requires counting in the model space.

1) BOOLEAN SATISFIABILITY

Let $g : \mathbb{B}^n \to \mathbb{B}$ be a Boolean function. Then, the ON-set of g is defined as ON-set$(g) = \{x \in \mathbb{B}^n | g(x) = 1\}$. The Boolean satisfiability problem is formally defined as follows. Given a Boolean function g, decide whether ON-set$(g) \neq \emptyset$ and if this is the case, determine $a \in$ ON-set(g). In other words, the goal is to find an input assignment a (so-called model) for which $g(a) = 1$ or inform us that no such a exists. Such an assignment is called satisfiable assignment. Note that if the ON-set contains more satisfiable assignments, one of them is returned.

Definition 13: Let SATOne(g) denote a procedure returning a single satisfiable assignment defined as

$$\mathrm{SATOne}(g) = \begin{cases} a \in \text{ON-set}(g) & \text{iff } \text{ON-set}(g) \neq \varnothing \\ \varnothing & \text{otherwise,} \end{cases} \tag{13}$$

where g is a single output Boolean function $g : \mathbb{B}^n \rightarrow \mathbb{B}$. In order to keep the algorithms simple, we will use SATOne(g) also as a predicate P in conditional expressions with the following meaning $P = \mathrm{SATOne}(g) \neq \varnothing$, i.e. the predicate is true iff the ON-set(g) is not empty.

According to the Cook-Levin theorem, the computational complexity of the Boolean satisfiability problem and hence the SATOne operation is NP-complete [36]. Despite this, the use of the SAT problem has been investigated in the field of digital system design for more than twenty years and many powerful tools utilizing SAT solvers have been developed. Modern SAT algorithms are very effective at coping with large problem instances and large search spaces [37].

2) LEXICOGRAPHIC BOOLEAN SATISFIABILITY

The lexicographic Boolean satisfiability (LEXSAT) problem is a modified version of the Boolean satisfiability problem. Compared to the Boolean satisfiability, LEXSAT finds the lexicographically smallest satisfying assignment (i.e. an assignment whose integer value under the given variable order is minimum among all satisfiable assignments). If the formula has no satisfying assignments, it proves it unsatisfiable, as does the traditional SAT. Formally, the LEXSAT decision procedure finds an input assignment a for which $g(a) = 1$ such that $g(b) = 0$ for all $b < a$. Then, a is called the lexicographically smallest assignment. If no such assignment exists, it returns $\varnothing$.

The LEXSAT is NP-hard and complete for the class $\mathsf{FP}^{\mathsf{NP}}$ (i.e. for the class of functions computable in polynomial time with the polynomial number of queries to an NP oracle [38]). At most n calls of a SATOne procedure are required to refine an assignment to the input variables to be the lexicographically smallest for an n-input Boolean function [39].

3) MODEL COUNTING

Another operation closely related to satisfiability is model counting (sometimes also referred to as SAT counting) which computes the number of input assignments a for which $g(a) = 1$ (i.e. it determines the number of satisfiable assignments for a given formula).

Definition 14: Let SATCount(g) denote a procedure returning the number of satisfiable assignments defined as

$$\mathrm{SATCount}(g) = |\,\text{ON-set}(g)| = \sum_{\forall x \in B^n} [\![g(x) = 1]\!] \\ = \sum_{\forall x \in B^n} g(x), \tag{14}$$

where $g : \mathbb{B}^n \rightarrow \mathbb{B}$.

The model counting generalizes SAT and its computational complexity is known to be #P-complete [36]. It represents a challenging problem since it has been demonstrated that #SAT is extremely hard even for some polynomial-time solvable problems [40]. In other words, it is a well known fact that there exists a class of relatively easy decision problems whose counting version is hard.

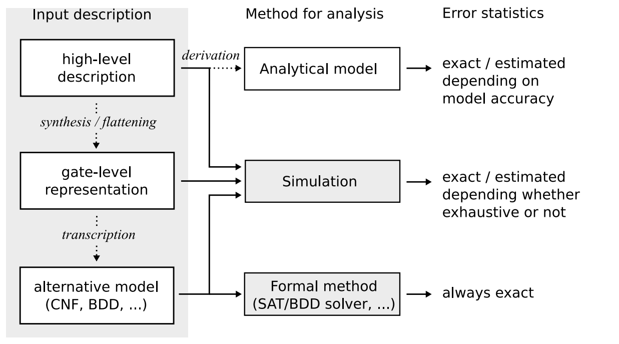

Input description
Method for analysis
Error statistics
high-level description
synthesis / flattening
gate-level representation
transcription
alternative model (CNF, BDD, ...)
derivation
Analytical model
Simulation
Formal method (SAT/BDD solver, ...)
exact / estimated depending on model accuracy
exact / estimated depending whether exhaustive or not
always exact

FIGURE 2. Methods for analysis of approximate circuits functionality and their accuracy.

IV. ERROR METRICS COMPUTATION

In this section, algorithms for computation of the error metrics mathematically defined in the previous section are presented. We show the way to formulate the error computation at the abstract level using either SATOne or SATCount operation. The obtained algorithms can be then understood as a universal recipe on how to determine the error, regardless of the chosen formal apparatus.

This section is divided into two parts. The first part provides a more detailed insight into the formal techniques. It discusses their properties, scalability, applicability and especially the way to implement SATOne and SATCount operations.

A. METHODS FOR COMPUTATION OF ERROR METRICS

The methods allowing for the accurate analysis of approximate circuits and their relation to the chosen representation of the approximate circuits are summarized in Figure 2. The analytical approaches typically profit from the structural description of the approximate circuits and are applicable to those circuits that exhibit a regular structure. Compared to other approaches, analytical methods require to derive the analytical model. This can be done either manually, semi-manually or automatically by means of a theorem prover as shown for example in [41]. The simulation can be performed independently of the circuit structure and representation. The chosen representation, however, affects the duration of the simulation. Similarly, the formal techniques can be applied to circuits, regardless of the level of description, but they require to transform the problem to a more suitable representation. SAT solvers insist on Conjunctive Normal Form (CNF). BDD packages represent the circuits using reduced ordered BDDs (ROBDDs). Symbolic computation approaches require to specify the problem using algebraic equations [18].

If formal guarantees on the error have to be ensured, we have to employ either the exhaustive simulation or a

TABLE 2. **Properties of the methods for accurate analysis of approximate circuits.**

	method			
property	*exhaustive simulation*	*binary decision diagrams*	*satisfiability solvers*	*#sat solvers*
arbitrary error metric	yes	no	no	no
worst-case analysis	yes	yes	yes	no
average-case analysis	yes	yes	no	yes
approximation miter	optional	required	required	required
computational complexity – best case	exponential	polynomial	polynomial	polynomial
computational complexity – worst case	exponential	polynomial	exponential	exponential
memory requirements – best case	low	moderate	moderate	moderate
memory requirements – worst case	low	exponential	moderate	high
applicability	typically ≤ 32 PIs	circuit dependent	circuit dependent	circuit dependent

formal approach like BDDs or SAT solvers. Each method, however, behaves differently. The properties of these methods are summarized in Table 2 and further discussed in the following paragraphs.

1) EXHAUSTIVE SIMULATION

The circuit simulation represents the most universal method as it can be used to evaluate virtually arbitrary error metric. The circuit simulator is used to determine the output value $\hat{f}(x)$ for a given input vector x. The remaining computation is done in the software. It is not even necessary to synthesize the reference circuit to determine $f(x)$. The output value of the exact circuit can be computed on-the-fly using a high-level model[1] of the reference circuit. In order to maximize performance, bit-parallel circuit simulation is typically used. The parallel simulation utilizes parallel architecture of modern processors by simulating the circuit on multiple input vectors in a single pass through the circuit. It profits from the fact that standard CPU today supports bitwise operations, at least on 64-bit integers. Considering this fact, it is beneficial to transform the simulated circuit to the equivalent gate-level representation. The modern CPUs, equipped with AVX2, support even 256-bit operations. Moreover, 512-bit operations have been recently introduced in AVX-512 extension. Despite that, the exhaustive simulation scales poorly.

The computational time grows exponentially with the increasing number of primary inputs and linearly with the number of gates of a simulated circuit. The exhaustive simulation is thus tractable for circuits having up to 32 inputs. This corresponds with $n = 2^{32} = 4.3 \times 10^9$ input vectors that can be evaluated in $n/64 = 67.1 \times 10^6$ passes when 64-bit instructions are employed.

2) BDDS, SAT AND #SAT SOLVERS

In order to apply the formal methods, we need to transform the problem into a Boolean satisfiability problem. A single output circuit denoted as miter is typically

[1]Note that the model is reduced to a common addition, multiplication or division operation for arithmetic circuits such as approximate adders, multipliers or dividers, respectively.

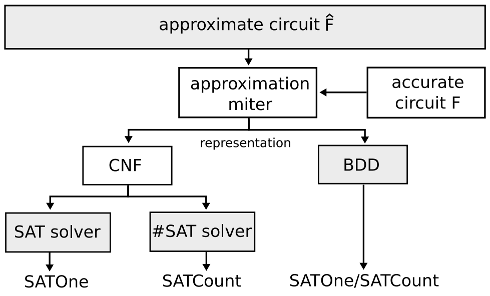


FIGURE 3. **Schematic diagram of the approximate circuit analysis using formal approaches.**

constructed to prove the functional equivalence of two circuits (see Figure 4a). The outputs of the circuit evaluate to one if and only if the analyzed circuits provide different outputs. When translated to CNF, a SAT solver can be used to check the equivalence. This principle can also be adopted for the purpose of approximate computing; however, we need to construct the so-called approximation miter (see Figure 4 and 5) computing the difference between the accurate and approximate circuits (respecting the chosen measure). The miter is then represented as a CNF or ROBDD and further analyzed. This process is illustrated in Figure 3. Unfortunately, the necessity of constructing the approximation miter makes the computation of some metrics non-trivial. The definition of the mean relative error, for example, would require an approximation miter containing a divider with a rational output. Such a circuit is prohibitively complex to be effectively represented as ROBDD or solved using SAT solvers.

BDDs can be directly used for the worst-case, as well as the average-case, analysis because every library for ROBDD manipulation is typically equipped with SATCount as well as SATOne operation. Common SAT solvers are, in principle, applicable to the worst-case analysis only. The SAT solver receives one formula in CNF and decides whether it is satisfiable or unsatisfiable. This means that it performs the SATOne procedure only. The average-case error analysis, however, requires us to determine the number of input assignments that evaluates the output of an approximation miter to true.

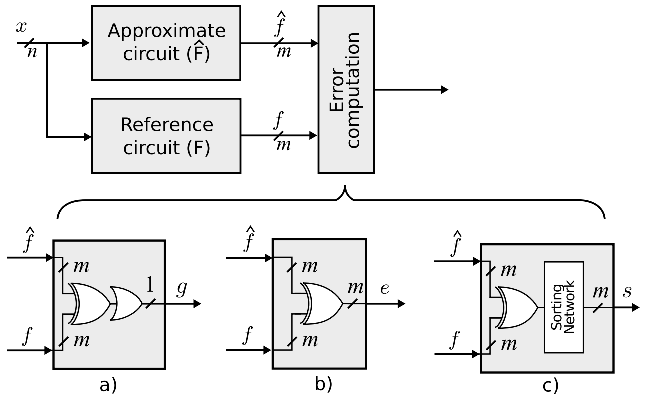


FIGURE 4. **Miter circuits for an analysis of a) error rate, b) average Hamming distance, and c) maximum Hamming distance.**

Hence, a special #SAT solver is required to address this task. Encoding the approximation miter as CNF expressed in terms of the input variables would be impractical due to the exponential increase in the formula size. Hence, a more convenient approach based on Tseitin transformation is typically applied in practice to obtain a CNF formula which can be solved by a SAT solver. The main advantage of this approach is that the CNF size grows linearly relative to the number of gates. For more details, please refer to [36].

Considering the computational complexity of the formal methods, the main advantage of ROBDDs is the possibility of efficiently performing many of the operations needed for the manipulation or querying of Boolean functions. The equivalence test of two Boolean functions, for example, can be done in constant time because almost every BDD package implements advanced node sharing. The checking is thus reduced to a comparison of pointers. A satisfying assignment can be computed in linear time with respect to the number of BDD variables. Determining the number of satisfiable assignments can be done in linear time with respect to the number of BDD nodes. Regarding SAT solvers, their speed mainly depends on the number of paths that have to be traversed in order to prove or disprove the satisfiability. Despite the compact CNF representation, the number of paths traversed by the SAT solver may grow exponentially by increasing the number of inputs. It means that even the common equivalence checking may represent a serious problem for SAT solvers when applied to some pathological cases like multipliers and dividers.

The memory requirements follow the opposite trend. The SAT solvers usually have moderate memory requirements. The decision procedure is based on a sophisticated backtracking [40]. The underlying data structures that ensure good scalability are relatively small. On the contrary, it is the requirement of canonicity which makes BDDs inefficient in representing certain classes of functions. For example, multipliers are known for their exponential memory requirements for any variable ordering [42]. It was shown in [43] that the BDD for the multiplier of two n-bit numbers has at least $2^{n/8}$ nodes. It is also a well-known fact that the size of a BDD (i.e. the number of non-terminal nodes) for a given function is very sensitive to the chosen variable order. Depending on the actual variable order, there are Boolean functions for which the size of the ROBDD can be either linear or exponential in the number of nodes [44].

The fact that it is not possible to perform equivalence checking of common 32-bit (exact) multipliers does not mean that the SAT solvers cannot be used in the context of approximate computing. Ceska et al., for example, proposed a design method capable of approximating even 32-bit multipliers without sacrificing the requirement for formal error guarantees [16].

B. ERROR PROBABILITY

The procedure for determining the error probability is relatively straightforward as it can be based on a miter routinely used to prove the functional equivalence. The miter (see Figure 4a) contains the combinational circuits whose corresponding outputs are connected via XOR gates whose outputs are fed into a single OR gate. It means that it evaluates to true if and only if a certain input assignment results in a difference of outputs. To disprove functional equivalence, it is sufficient to find at least one input assignment that evaluates to true. For this purpose, the SATOne operation can be employed.

The error probability defined as the fraction of input vectors for which the approximate output differs from the original one, can be determined using the SATCount operation as follows:

$$\begin{aligned} e_{prob}(f,\hat{f}) &= \frac{1}{2^n}\sum_{\forall x\in\mathbb{B}^n}[\![f(x)\neq\hat{f}(x)]\!] \\ &= \frac{1}{2^n}\sum_{\forall x\in\mathbb{B}^n}\left(\bigvee_{0\le i<m} f_i(x)\oplus\hat{f}_i(x)\right) \\ &= \frac{1}{2^n}\,\mathrm{SATCount}\left(\bigvee_{0\le i<m} f_i\oplus\hat{f}_i\right) \end{aligned} \tag{15}$$

In Eq. 15, we start with the definition of error rate (Eq. 2), then the Iverson bracket is substituted (Eq. 1) and finally, the sum is replaced according to the definition of SATCount (Eq. 14). The expression inside the SATCount operation does not depend on x and can thus be represented as a single-output Boolean circuit whose output corresponds with signal g in Figure 4a. This circuit represents the approximation miter that can either be converted to CNF and submitted to a SAT solver or represented as ROBDD as shown in Figure 3.

C. AVERAGE HAMMING DISTANCE

The principle of the average-case Hamming distance computation is analogical. The average-case Hamming distance is obtained by using the same miter, but without the final OR gate (see Figure 4b). Then, the SATcount operation is called for each XOR output. Finally, the obtained results are summed and divided by the total number of input assignments. This algorithm is obtained when we transform Eq. 2

into an equivalent one as follows:

$$
\begin{aligned}
e_{mhd}(f,\hat{f}) &= \frac{1}{2^n}\sum_{\forall x\in B^n}\left(\sum_{i=0}^{m-1} f_i(x)\oplus\hat{f}_i(x)\right)\\
&= \frac{1}{2^n}\sum_{i=0}^{m-1}\left(\sum_{\forall x\in B^n} f_i(x)\oplus\hat{f}_i(x)\right)\\
&= \frac{1}{2^n}\sum_{i=0}^{m-1}\text{SATCount}(f_i\oplus\hat{f}_i). \qquad (16)
\end{aligned}
$$

The Hamming distance computed using BDDs was introduced in the context of approximate synthesis of general logic in [45] and later in [13].

D. WORST-CASE HAMMING DISTANCE

This metric is rarely used in the literature. It has been applied in the context of stochastic computing [32] but without introducing a formal approach for its computation. One approach is to follow Eq. 4 and construct a miter which consists of the XOR gates whose outputs are summed by a tree of adders. Then, we can use an algorithm for determining the worst-case value as shown in Section IV-G. Another possibility on how to determine the worst-case Hamming distance is to use a sorting network instead of the adder (see Figure 4c). The sorting network is a fixed structure of comparators used to reorder an arbitrary data sequence. The main benefit of the bit-level sorting network is that each comparator consists of a single AND and single OR gate. The AND gate computes the minimum value and OR gate is used to compute the maximum value. Such a structure should be more suitable for BDDs/SAT solvers compared to the usage of an adder tree since it does not contain XOR chains. Let $\text{SN}^{(k)} : \mathbb{B}^k \rightarrow \mathbb{B}^k$ be a sorting network which sorts all existing binary inputs from $\mathbb{B}^k$ into a descending sequence of binary values. Then, the bit-flip error is determined as follows:

$$
e_{bf}(f,\hat{f}) = \arg\max_{z}\left\{\text{SATOne}\left(\text{SN}_z^{(m)}(f\oplus\hat{f})\right) = \text{true}\right\}. \qquad (17)
$$

We are looking for an input assignment x which activates the maximum number of ones in $f \oplus \hat{f}$. In other words, we are looking for the most significant output (i.e. leftmost) of the sorting network SN which evaluates to one. This procedure is formalized in Algorithm 1 which receives the approximation miter and computes e_{bf}. The approximation miter corresponds with the m-output Boolean function $s = \text{SN}^{(m)}(f \oplus \hat{f})$. A binary search is used to identify the leftmost active output of s.

The algorithm begins by checking whether there exists an input assignment γ that evaluates the middle output of the SN (denoted as s_z) to one. If such γ exists, the search continues in the upper half of the SN outputs. Otherwise, the search continues in the lower half of the outputs. The algorithm furthermore demonstrates the way of employing the knowledge of γ in order to improve the efficiency of the search (see line 6 and 7). We can determine the outputs of

Algorithm 1 Worst-Case Hamming Distance Analysis (Binary Search)

```
Input: approximation miter with m-bit sorting network
       SN^(m) (s)
Output: maximum Hamming distance ($e_{bf}$)
1  $l \leftarrow 0; r \leftarrow m-1$
2  while $l \le r$ do
3      $z \leftarrow \lceil (l+r)/2 \rceil$
4      if ($\gamma \leftarrow$ SATOne($s_z$)) $\neq \varnothing$ then
5          $l \leftarrow z+1$
           // γ is used to narrow the interval:
6          while $(l \le r) \wedge s_l(\gamma)$ do
7              $l \leftarrow l+1$
8      else
9          $r \leftarrow z-1$
10 return $l$
```

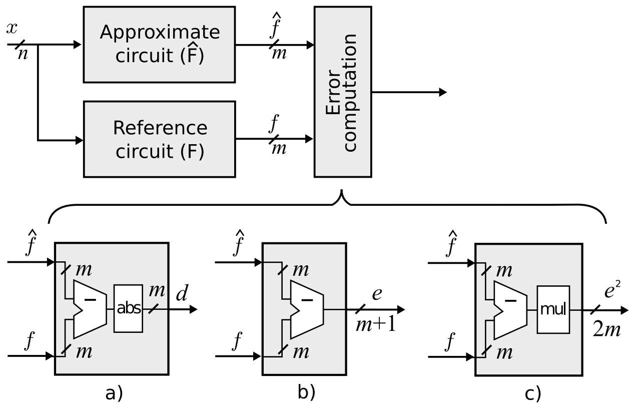


FIGURE 5. Miter circuits for analysis of arithmetic circuits. Miter producing a) absolute error distance, b) error distance, and c) squared error distance (naïve approach).

SN that are set to one and safely skip them because we have already found the input assignment that evaluates them to one. The value of the l-th output of SN for a particular input assignment γ is denoted as $s_l(\gamma)$. Compared to the BDD-based analysis, the computation of $s_l(\gamma)$ is for free if the SAT solver is employed because every SAT solver returns the state of all intermediate variables including $s_l(\gamma)$. Each variable can be in one of three states – it can be assigned to true, false, or undefined. In the algorithm, we are interested in those variables that are in the 'true' state. In particular, we are looking for the highest l for that $s_l(\gamma) = \mathit{true}$.

E. MEAN ABSOLUTE ERROR

In order to calculate the mean absolute error, an approximation miter determining the arithmetic error distance has to be constructed. The miter typically consists of the exact as well as approximate circuits whose outputs are fed into the subtracter followed by a circuit which computes the absolute value (see Figure 5a). Subtraction can be calculated using $m+1$ full-adders with first carry-in set to 1 and inverting each bit of the subtrahend. The absolute value can be implemented using m half-adders and m XOR gates.

There are several methods how to determine the mean absolute error. For example, an analysis based on the construction of a characteristic function was proposed in [13]. In order to avoid the building of a characteristic function, which may be time-consuming, an alternative approach was proposed in [14]. It exploits the fact that there exists a more convenient form of Eq. 7 which can be employed in practice. Let $D(x) = |\mathrm{nat}(f(x)) - \mathrm{nat}(\hat{f}(x))|$ be the output of the approximation miter and $d = \mathrm{nat}^{-1}(D)$ its m-bit binary representation. Considering the structure of the approximation miter, d are the output bits of circuit determining the absolute value. The equation for error analysis is constructed as follows:

$$\begin{aligned} e_{mae}(f,\hat{f}) &= \frac{1}{2^n}\sum_{\forall x\in\mathbb{B}^n} D(x) = \frac{1}{2^n}\sum_{\forall x\in\mathbb{B}^n}\left(\sum_{i=0}^{m-1}2^i d_i(x)\right) \\ &= \frac{1}{2^n}\sum_{i=0}^{m-1}\left(2^i\sum_{\forall x\in\mathbb{B}^n} d_i(x)\right) \\ &= \sum_{i=0}^{m-1}2^{i-n}\cdot\mathrm{SATCount}(d_i). \end{aligned} \tag{18}$$

The mean absolute error is obtained by m calls of SATCount operation, one per each output bit. The obtained counts are weighted according to the significance of the output bits and summed up.

1) THE PROPOSED ALGORITHM

To further improve efficiency, we propose to directly use the output of the subtracter. The subtracter outputs a signed value $E(x) = \mathrm{nat}(f(x)) - \mathrm{nat}(\hat{f}(x))$ represented using $(m+1)$-bits. Let $e = \mathrm{nat}^{-1}(E)$ be the binary representation of E. For a common subtracter representing numbers in two's complement it holds that $E = -2^m e_m + \sum_{i=0}^{m-1} 2^i e_i$. The absolute value $|E|$ can thus be expressed as

$$|E| = \begin{cases} \sum_{i=0}^{m-1} 2^i e_i, & \text{when } e_m = 0 \\ -\left(-2^m + \sum_{i=0}^{m-1} 2^i e_i\right), & \text{when } e_m = 1. \end{cases} \tag{19}$$

We can then substitute $|E|$ into Eq. 7. The computation consequently breaks down into two parts – summing $E(x)$ for all positive cases and summing $-E(x)$ for all negative cases. The positiveness or negativeness of $E(x)$ is determined by the sign bit, i.e. $e_m(x)$. To restrict SATCount(g) to positive cases only, it is sufficient to condition g by $\overline{e_m}$. Similarly, the condition $g \wedge e_m$ restricts the SAT counting to negative cases only. This principle is used to formulate the computation of the mean absolute error as $2m+1$ calls of the SATCount operation as shown in Eq. 20.

$$\begin{aligned} &e^{II}_{mae}(f,\hat{f}) \\ &= \frac{1}{2^n}\sum_{\forall x\in\mathbb{B}^n}|E(x)| \\ &= \frac{1}{2^n}\Bigg[\sum_{\substack{\forall x\in\mathbb{B}^n\\ e_m(x)=0}} E(x) + \sum_{\substack{\forall x\in\mathbb{B}^n\\ e_m(x)=1}} -E(x)\Bigg] \\ &= \frac{1}{2^n}\Bigg[\sum_{\substack{\forall x\in\mathbb{B}^n\\ e_m(x)=0}}\left(\sum_{i=0}^{m-1}2^i e_i(x)\right) + \sum_{\substack{\forall x\in\mathbb{B}^n\\ e_m(x)=1}}\left(2^m - \sum_{i=0}^{m-1}2^i e_i(x)\right)\Bigg] \\ &= 2^{m-n}\,\mathrm{SATCount}(e_m) \\ &\quad + \sum_{i=0}^{m-1}2^{i-n}\Big(\mathrm{SATCount}(\overline{e_m}\wedge e_i) - \mathrm{SATCount}(e_m \wedge e_i)\Big) \end{aligned} \tag{20}$$

The obtained equation looks complicated compared to Eq. 18, but it helps us to avoid the usage of the XOR-intensive absolute value in the miter. In addition, we can introduce some optimizations. For example, $\mathrm{SATCount}(e_m) = 0$ implies that $\mathrm{SATCount}(e_m \wedge e_i) = 0$ and all computations of this term can thus be safely skipped. In other words, if the difference is always positive, we can omit the part related to the negative values.

Note that $2^m - \sum_{i=0}^{m-1} 2^i e_i$ equals to the $1 + \sum_{i=0}^{m-1} 2^i \overline{e_i}$. This equality can be used to derive another alternative formula:

$$\begin{aligned} &e^{III}_{mae}(f,\hat{f}) \\ &= \frac{1}{2^n}\Bigg[\sum_{\substack{\forall x\in\mathbb{B}^n\\ e_m(x)=0}}\left(\sum_{i=0}^{m-1}2^i e_i(x)\right) + \sum_{\substack{\forall x\in\mathbb{B}^n\\ e_m(x)=1}} 1 \\ &\quad + \sum_{\substack{\forall x\in\mathbb{B}^n\\ e_m(x)=1}}\left(\sum_{i=0}^{m-1}2^i \overline{e_i}(x)\right)\Bigg] = \frac{1}{2^n}\mathrm{SATCount}(e_m) \\ &\quad + \sum_{i=0}^{m-1}2^{i-n}\Big(\mathrm{SATCount}(\overline{e_m}\wedge e_i) + \mathrm{SATCount}(e_m\wedge \overline{e_i})\Big) \\ &= 2^{-n}\,\mathrm{SATCount}(e_m) + \sum_{i=0}^{m-1}2^{i-n}\,\mathrm{SATCount}(e_m \oplus e_i) \end{aligned} \tag{21}$$

The computation of the absolute value which requires additions and XORing with the sign bit was moved from the circuit level to the algorithm level. As in the previous case, $\mathrm{SATCount}(e_m) = 0$ implies that e_m is always zero which means that the Boolean function $e_m \oplus e_i$ is equal to e_i. This procedure is formalized as Algorithm 2.

F. MEAN SQUARED ERROR

A naïve approach on how to determine the mean squared error is to use a multiplier whose both operands are connected to the output of the subtracter as shown in Figure 5c. The obtained miter, however, leads to an extremely inefficient computation since the squaring is done at the level of a circuit using a large array of one-bit full-adders. In addition, the squaring implies that the partial products will contain common terms that have to be summed twice.

In a more efficient procedure and similar to the mean absolute error calculation, we start with the expansion of

Algorithm 2 Computation of the Mean Absolute Error

```
Input: approximation miter with signed output ($e$)
Output: mean absolute arithmetic error ($e_{mae}$)
1  $c \leftarrow$ SATCount($e_m$); $\varepsilon \leftarrow c$
2  for $i \in \{0, 1, \ldots, m-1\}$ do
3      if $c > 0$ then
4          $\varepsilon \leftarrow \varepsilon + 2^i$ SATCount($e_i \oplus e_m$)
5      else
6          $\varepsilon \leftarrow \varepsilon + 2^i$ SATCount($e_i$)
7  return $2^{-n}\varepsilon$;
```

equation for expression E^2. It holds that

$$E^2 = \left(-2^m + \sum_{i=0}^{m-1} 2^i e_i\right)^2$$
$$= \sum_{i=0}^{m} 2^{2i} e_i + \sum_{\substack{i,j=0\\ j>i}}^{m-1} 2^{1+i+j} e_i e_j - \sum_{i=0}^{m-1} 2^{1+i+m} e_i e_m \qquad (22)$$

Then, we can substitute this expression into Eq. 8 and after rearranging it we obtain the following form:

$$e^I_{mse}(f, \hat{f})$$
$$= \frac{1}{2^n} \sum_{\forall x \in \mathbb{B}^n} (E(x))^2$$
$$= \frac{1}{2^n}\left[\sum_{i=0}^{m} 2^{2i}\,\text{SATCount}(e_i) - \sum_{i=0}^{m-1} 2^{i+m+1}\text{SATCount}(e_i \wedge e_m)\right.$$
$$\left. + \sum_{\substack{i,j=0\\ j>i}}^{m-1} 2^{1+i+j}\,\text{SATCount}(e_i \wedge e_j)\right] \qquad (23)$$

An optimized variant of the algorithm computing the mean squared error, as derived in Eq. 23, is provided as Algorithm 3.

Algorithm 3 Computation of the Mean Squared Error

```
Input: approximation miter with signed output ($e$)
Output: mean squared arithmetic error ($e_{mse}$)
1  $\varepsilon \leftarrow 0$
2  for $i \in \{0, 1, \ldots, m\}$ do
3      $c \leftarrow$ SATCount($e_i$); $\varepsilon \leftarrow \varepsilon + 2^{2i}c$
4      if $c > 0$ then
5          for $j \in \{i+1, \ldots, m\}$ do
6              $c \leftarrow$ SATCount($e_i \wedge e_j$)
7              if $j = m$ then
8                  $c \leftarrow -c$
9              $\varepsilon \leftarrow \varepsilon + 2^{i+j+1}c$
10 return $2^{-n}\varepsilon$;
```

The algorithm starts with the computation of the number of satisfiable assignment for e_i. If there is no such assignment, we can skip the rest of the code as it depends on e_i which is never evaluated to true. The inner loop implements the second and third sum of Eq. 23. The second sum is the case when $j = m$.

G. WORST-CASE ABSOLUTE ERROR

The literature distinguishes two different problems related to the worst-case error: worst-case error analysis and worst-case error checking. The output of the worst-case error analysis algorithm is the maximum error magnitude observed on the output of the analyzed approximate circuit. The goal of the checking is to prove whether the worst-case error is less or greater than a chosen level. The latter problem is much simpler compared to the worst-case error analysis as it can be computed using a single call of SATOne. The experimental results indicate that the real computational complexity should not be worse than the computational complexity of common equivalence checking, since even 32-bit multipliers have been successfully analyzed within a few seconds [16].

1) WORST-CASE ERROR CHECKING

To compute whether the ED is violated, we can use the approximation miter, as shown in Figure 5b, followed by a decision block which checks whether the error introduced by the approximation is greater than a given threshold. For a fixed threshold, the decision block can be implemented efficiently using simple AND / OR gates as proposed in [16]. A comparison of signal A with a constant bit vector T representing the threshold expressed on k bits is substituted by the following logic expression:

$$A > T \equiv \bigvee_{\substack{0 \le i \le k-1\\ T_i = 0}} \left(A_i \wedge \bigwedge_{\substack{i < j \le k-1\\ T_j = 1}} A_j\right). \qquad (24)$$

For a given threshold $\mathcal{T}$ on the worst-case absolute error, it holds that $e_{wce} > \mathcal{T}$ is satisfied if either the output of subtracter e is positive and $e > \mathcal{T}$, or e is negative and $-e > \mathcal{T}$. As we typically deal with numbers in the two's complement, the second condition is equal to $\bar{e} > (\mathcal{T} - 1)$. Hence, we can use the two's complement representation and examine the positive and negative values separately to avoid the usage of the absolute difference of the output. Formally:

$$\text{WCEGT}(E, \mathcal{T})$$
$$= \exists_{x \in \mathbb{B}^n} |E(x)| > \mathcal{T}$$
$$= \exists_{x \in \mathbb{B}^n}\left[(E(x) \ge 0 \wedge E(x) > \mathcal{T}) \vee (E(x) < 0 \wedge -E(x) > \mathcal{T})\right]$$
$$= \text{SATOne}\Big([\overline{e_m} \wedge (E > \mathcal{T})] \vee [e_m \wedge (\overline{E} > (\mathcal{T} - 1))]\Big) \neq \varnothing, \qquad (25)$$

where $E(x) = \text{nat}(f(x)) - \text{nat}(\hat{f}(x))$.

2) WORST-CASE ERROR COMPUTATION

An iterative algorithm for the worst-case error analysis computing the maximum value of the difference between the original and approximate circuit suitable for BDDs was presented in [13], [14]. It requires us to construct the approximation miter producing the absolute difference D as shown in Figure 5a. The algorithm iterates over the output bits of $D(x) = \sum_{0 \le i < m} d_i(x) \cdot 2^i$. It starts with the most significant bit d_{m-1} and checks whether there exists an input assignment that makes this bit active. Input assignments that do not evaluate to the maximum value are ruled out using the mask μ (please refer to [13] for more details).

A pseudocode of an improved variant of the algorithm published in [13] is shown as Algorithm 4. Compared to [13], our version analyzes the model returned by SATOne operation. It tries to determine the next bits without the necessity of checking them explicitly. We employ a similar principle as discussed in Algorithm 1.

Algorithm 4 worst-Case Error Analysis (Optimized Sequential Approach for Miter With Unsigned output)

```
Input: approximation miter with absolute value circuit
       giving the unsigned output ($d$)
Output: maximum absolute arithmetic error ($e_{wce}$)
1  $\varepsilon \leftarrow 0, \mu \leftarrow true$
2  for $i \in \{m-1, m-2, \dots, 0\}$ do
3      if ($\gamma \leftarrow$ SATOne($\mu \wedge d_i$)) $\neq \varnothing$ then
4          $\mu \leftarrow \mu \wedge d_i$
5          $\varepsilon \leftarrow \varepsilon + 2^i$
           // $\gamma$ is used to determine value of the
              next bits
6          while $(i > 0) \wedge d_{i-1}(\gamma)$ do
7              $i \leftarrow i - 1$
8              $\mu \leftarrow \mu \wedge d_i$
9              $\varepsilon \leftarrow \varepsilon + 2^i$
10 return $\varepsilon$;
```

As evident from Algorithm 4, we require a suitable formal approach that allows us to modify the approximation miter during the computation, ideally without losing the information gathered in the previous steps. The BDD packages have this ability as the BDD nodes are cached in memory. When a SAT solver is used, however, the CNF formulas are typically solved separately. Fortunately, the modern SAT solvers support the incremental SAT solving mode allowing them to propagate the information gathered during the solving process to future instances. The incremental SAT solver enables us to append an additional CNF clause, but the most efficient way on how to implement the mask μ is to use the so-called assumptions (propositions that hold solely for one specific invocation of the solver) [46]. In our case, the assumptions correspond with μ because at least one additional condition is added to μ in each iteration.

Algorithm 4 requires a miter producing the absolute difference. There are two possibilities on how to modify the procedure to directly use the signed output of the subtracter. One possibility is to separately check the maximum positive and maximum negative error. Another possibility is to emulate

Algorithm 5 worst-Case Error Analysis (Optimized Sequential Approach for Miter With Signed output)

```
Input: approximation miter with signed output ($e$)
Output: maximum absolute arithmetic error ($e_{wce}$)
1  $\varepsilon \leftarrow 0, \mu \leftarrow true, sgn \leftarrow$ SATOne($e_m$) $\neq \varnothing$
2  $d \leftarrow e \oplus e_m$ if $sgn$ else $e$
3  for $i \in \{m-1, m-2, \dots, 0\}$ do
4      if ($\gamma \leftarrow$ SATOne($\mu \wedge d_i$)) $\neq \varnothing$ then
5          $\mu \leftarrow \mu \wedge d_i$
6          $\varepsilon \leftarrow \varepsilon + 2^i$
           // $\gamma$ is used to determine value of the
              next bits
7          while $(i > 0) \wedge d_{i-1}(\gamma)$ do
8              $i \leftarrow i - 1$
9              $\mu \leftarrow \mu \wedge d_i$
10             $\varepsilon \leftarrow \varepsilon + 2^i$
   // recent $\gamma$ is used to determine value of the
      sign bit and avoid calling SATOne
11 if $d_m(\gamma) \vee$ SATOne($\mu \wedge e_m$) then
12     $\varepsilon \leftarrow \varepsilon + 1$
13 return $\varepsilon$;
```

the absolute difference at the level of the algorithm as it was applied in Algorithm 2. A pseudocode corresponding with the latter case is shown in Algorithm 5.

Firstly, the output values that have the signed bit active are inverted using XOR operation (line 2). Here we treat the output value as a number represented in the ones' complement because the ones' complement behaves like the negative of the original number to within a constant of -1. Then, exactly the same approach as used in Algorithm 4 is applied to determine the maximum value of non-negative d. Finally, a correction is made if necessary (line 12) to fix possibly incorrect value caused by the usage of the ones' complement. The correction is necessary only if the maximum value of d is coming from negative e. The model returned by a SAT solver is used to improve the performance as shown in Algorithm 4. In addition, the model is used in the correction phase to avoid an unnecessary SATOne call.

Algorithm 6 worst-Case Error Analysis (Binary Search)

```
Input: approximation miter with signed output ($e$)
Output: maximum arithmetic error ($e_{wce}$)
1  $l \leftarrow 0; r \leftarrow 2^m - 1$
2  while $l \le r$ do
3      $t \leftarrow \lceil (l + r)/2 \rceil$
4      if WCEGT($e, t$) then
5          $l \leftarrow t + 1$
6      else
7          $r \leftarrow t - 1$
8  return $l$
```

Finally, let us conclude this part with the algorithm based on the binary search shown as Algorithm 6. The principle of this procedure is to iteratively check whether the error is greater than a given threshold (denoted as t in the algorithm). The search procedure gradually narrows down the interval where the exact error value lies. After a finite number of steps, a single value is determined. As the binary search runs in logarithmic time with respect to the range, at most m comparisons are required. The checking can be ensured by means of the magnitude comparator as shown in Eq. 25. An incremental SAT solver should be employed to mitigate a potential overhead caused by the necessity of constructing a different comparator in each iteration.

All three algorithms presented in this section can be considered as an implementation of a special form of LEXSAT solver. Instead of finding the lexicographically maximum input assignment, we are looking for the lexicographically maximum assignment to the variables associated with the output bits. The benefit coming with the use of SAT solvers compared to BDDs is that potentially less than m SAT calls are required to determine the worst-case value for an m-bit output because the solver can learn some bits from the received SAT assignments.

H. WORST-CASE RELATIVE ERROR

Although the definition of the worst-case relative error (Eq. 9) contains division, we will outline the way of determining the relative error and avoiding the use of a divider in the approximation miter.

Let us firstly show, how to compute whether the RED is violated using a subtracter and a constant multiplier:

$$\begin{aligned}\mathrm{WCREGT}(E,\mathcal{T}) &= \exists_{\substack{x\in\mathbb{B}^n\\ f(x)\neq 0}} \frac{|E(x)|}{nat(f(x))} > \mathcal{T}\\ &= \exists_{\substack{x\in\mathbb{B}^n\\ f(x)\neq 0}} |E(x)| > \mathcal{T}nat(f(x))\\ &= \exists_{\substack{x\in\mathbb{B}^n\\ f(x)\neq 0}} 2^k|E(x)| - 2^k\mathcal{T}nat(f(x)) > 0 \quad (26)\end{aligned}$$

For a given threshold $\mathcal{T} \in \mathbb{R}$, we are looking for an input assignment x which causes the absolute arithmetic error $E(x) = \mathrm{nat}(f(x)) - \mathrm{nat}(\hat{f}(x))$ to be greater than the value of the output of the correct circuit, i.e. $nat(f(x))$, multiplied by $\mathcal{T}$. Since the threshold $\mathcal{T}$ is a real number, we need to scale $E(x)$ as well as the results of the multiplication by a factor $k \geq 0$ (i.e. a constant which allows to represent $\mathcal{T}$ as integer). The multiplication by the constant $2^k\mathcal{T}$ can be done "multiplierless" using additions, subtractions, and binary shifts. The problem of finding the decomposition of a fixed point value $\mathcal{T}$ with the least number of operations is known as the single constant multiplication problem [47]. The absolute value can be replaced by its expanded form as shown in Eq. 25. The computation of WCREGT then leads to a single call of SATOne operation.

Once we have the WCREGT operation, we can adopt Algorithm 6 to determine the worst-case relative error. The only difference is that WCREGT has to be used instead of WCEGT. The bounds (i.e. the value of l and r) remain the same.

V. EXPERIMENTAL EVALUATION

We have implemented the algorithms described in Section IV in C++. In particular, we created a new module for the ABC tool [48] in order to have the possibility to work with Verilog files. Three new commands were created: axc_sim performing the error analysis based on exhaustive simulation, axc_bdd for analysis of approximate circuits using BDDs, and axc_sat which analyses the approximate circuits either by means of a SAT solver or #SAT solver. For BDD analysis, we utilized the BuDDY v2.4 [49] because our experience suggests that its performance is better than CUDD. We chose MiniSAT version 2.20 as the SAT solver supporting the incremental solving. For model counting, we chose sharpSAT by Thurley [50] representing the state-of-the-art solver for this problem. The exhaustive simulation is accomplished by the state-of-the-art parallel simulator utilizing common 64-bit bitwise instructions. Each algorithm is implemented as a single thread code to provide a fair comparison. The maximum allowed execution time is 3,600 seconds. Computations exceeding this limit are terminated. The initial number of BDD nodes was set to 10^8 which corresponds to 2 GiB of memory. The maximum amount of allocated memory was restricted to 6 GiB. The size of the caches used for the BDD operators was set to 10^7. All the other parameters were kept at the default values. The interleaved variable ordering optimal for adders was used for analysis of approximate adders, i.e. $a_{n-1} > b_{n-1} > a_{n-2} > b_{n-2} > \cdots > a_0 > b_0$, where a_i and b_i denote particular bits of the input operands. The least significant bit corresponds with the index 0. In other cases, a common ordering $a_{n-1} > a_{n-2} > \cdots > a_0 > b_{n-1} > b_{n-2} > \cdots > b_0$ was used.

A. BENCHMARKING METHODOLOGY

We conducted all experiments on a server equipped with two 2.2 GHz Intel Xeon E5-2630 CPUs, with 64GB LPDDR3 main memory each, running 64-bit Linux OS. Note that CPU affinity was forced and Hyper-Threading was disabled to maximize the accuracy. We ran each algorithm on a set of benchmark circuits. For each benchmark, we executed more runs, each with different reference circuit (the circuits are described in Section V-D). Rather than measuring the time of execution, which can be affected by other programs running on the same machine, and especially CPU frequency management, we employed the PAPI [51] interface which provides a high-level access to hardware performance counters. We observed the following hardware counters: the total number of instructions, the number of memory instructions, total CPU clock cycles, and CPU stall cycles. Non-stall CPU cycles converted to the time of execution are used to evaluate the performance of the algorithms. To calculate non-stall CPU cycles, we subtracted CPU stall cycles from the total clock cycles.

In order to understand the reported numbers, let us briefly recap how the performance of the error analysis is measured.

1) In the case of the exhaustive simulation, each gate-level netlist is first translated to a corresponding 64-bit machine code. The translation is done in linear time with respect to the number of gates. The obtained byte code is then executed and m 64-bit output values are produced. The values are rearranged to receive 64 integer values that are subsequently compared with 64 expected values to determine the contribution to the error parameters. The execution and comparison are repeated for the remaining input vectors. The input vectors are generated on the fly. The number of instructions and the CPU time reported in this paper takes into account the whole procedure, including the translation to the machine code. Note that the simulator computes all of the metrics simultaneously.
2) In the case of BDDs, the gate-level netlist is first converted to a corresponding BDD representation. The same conversion is done for the reference circuit specified by a command-line parameter. Then, the miter is finalized and the obtained ROBDD is analyzed and modified if necessary. The number of instructions and CPU time reported in this paper includes both phases (i.e. BDD construction as well as analysis). The construction of a BDD is typically computationally intensive and dominates the whole runtime.
3) In the case of SAT solving, the analyzed netlist as well as the requested reference circuit are firstly converted to a corresponding CNF using the Tseitin transformation. The conversion is done in linear time with respect to the number of gates. Finally, one or several calls of incremental SAT solver are executed. The number of instructions and CPU time for the initialization of the SAT solver, conversion to CNF and computation are reported in Section V-B.
4) The sharpSAT solver was extended to support the PAPI interface. Firstly, CNF is created as described in the previous paragraph. The CNF is then exported into a DIMACS format and sharpSAT is executed externally. The number of instructions and CPU time is measured since the SAT solver reads the DIMACs file.

Time and space complexity is typically considered in practice. We will focus strictly on the time complexity in the remainder of this work, partly because of requirements implied by the target application, partly because of the properties of formal approaches. As discussed in the introductory part, the error analysis needs to be done as quickly as possible because of its integration into an iterative design process. Considering this scenario, it means that the CPU time has a higher cost compared to the memory requirements. In addition, it makes no sense to compare the SAT solver-based implementations with the BDD-based ones. The implementations based on BDDs require substantially more memory (see Table 2). Finally, the amount of allocated memory for the BDD based implementations highly correlates with the time of computation because the construction phase dominates the whole time (see Section IV-A2).

TABLE 3. **Parameters of the approximate circuits utilized for benchmarking. Column bw shows the bit-width of the arithmetic circuits.**

circuit	bw	# instances	# gates			$e_{wce}\%$	
			min	max	median	min	max
adder	8	255	9	42	24	0	66
	12	345	13	64	37	0.01	69
	16	464	17	88	53	0.001	72
	32	239	76	246	146	0	0.2
multiplier	8	324	16	373	194	0.003	49
	12	159	29	857	294	0.001	20
	16	162	39	1675	471	0.001	20
	20	160	49	2667	740	0.001	20

B. BENCHMARKS

The performance of the error computation methods is evaluated on a large set of approximate circuits consisting of various approximate adders and multipliers. These arithmetic circuits were chosen because they represent key components of many real-world applications in signal processing and machine learning [1]. While the formal verification of adders represents a problem of moderate complexity, multipliers are known to be an extremely difficult benchmark and can be thus considered as a scalability indicator.

The approximate circuits are selected from the recent version of EvoApprox library which contains various arithmetic circuits created by means of evolutionary algorithms [16], [34] and more complex instances constructed from the smaller ones [12]. We have chosen the instances having a different design (power consumption, delay, area) as well as quality parameters. As it is assumed that the performance of the approximate error analysis depends not only on a type of circuit but also on the actual error, the instances are chosen to receive a representative sample which covers the whole range of values for each error parameter. In addition, we also included truncated multipliers and broken-array multipliers that exhibit a stable performance and achieve excellent design tradeoffs [12]. All circuits are available in the form of a gate-level Verilog netlist. Parameters of the benchmark circuits are summarized in Table 3.

In order to determine the key quality parameters, we performed a correlation analysis among the error parameters and key design parameters. The design parameters were obtained from the EvoApprox library. The resulting correlation heatmap, calculated over the 8-bit, 12-bit and 16-bit benchmarks (for which we have all of the relevant data) is shown in Figure 6. The normalized error metrics are used for a meaningful comparison across different bit widths. It is quite surprising to observe that many error parameters exhibit a close relationship. For example, e_{wce} strongly correlates with e_{mae} (Pearson correlation coefficient 1.0 for multipliers and

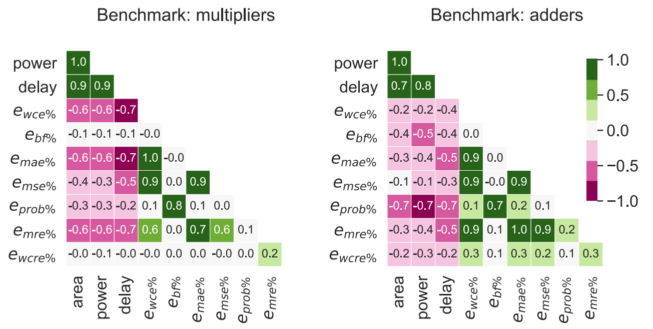


FIGURE 6. Correlation heatmap of design (area, power, delay) and quality parameters for the chosen benchmark circuits. The value -1 indicates a strong negative dependency, value 1 indicates a strong positive dependency.

0.9 for adders), e_{mse} (0.9) as well as e_{mre} (0.6 for multipliers, 0.4 for adders). Similarly, there is a positive correlation between error probability and bit-flip error (0.8 for multipliers, 0.7 for adders).

Looking at the design parameters (power, area, delay) and their correlation with error metrics, we can conclude that e_{wce} and e_{prob} represent a good candidate for indicating the quality of approximate multipliers and adders, respectively. Both classes of circuits exhibit a high negative correlation (around -0.7) among all design parameters and those metrics. It means that the power consumption/delay, as well as area, goes down if the error rate (the worst-case error for multipliers) increases.

C. COMPLEXITY AND SCALABILITY OF EXACT ERROR ANALYSIS OF APPROXIMATE CIRCUITS

The computational demand of the methods for error analysis of approximate circuits is summarized in Table 4. The table shows the averages calculated from all experiments. It means that several algorithms and different reference circuits are taken into account. Only the naïve algorithm for the mean squared error computation is intentionally omitted. The purpose of this evaluation is to assess the scalability of the formal methods and compare the computational requirements of each metric.

As expected, the exhaustive simulation is computationally intensive despite its simplicity and efficiency. Compared to the formal approaches, the number of issued instructions needed to evaluate 16-bit adders is about five orders of magnitude higher. On contrary, the main benefit of exhaustive simulation (i.e. its independence on the analyzed problem) is visible on multipliers where it performs better than SAT/BDDs for 8-bit and even some 12-bit instances. Due to its limited scalability, it is infeasible to conduct the exhaustive simulation for circuits having more than 16-bit operands.

The BDDs represent a quite powerful tool for analysis of the approximate adders and small multipliers. The computational complexity is relatively stable independently of the computed metric. The combinational equivalence checking and the analysis of the average Hamming distance can be

TABLE 4. The average number of instructions required to perform the error analysis. We report the median value calculated from all experiments in the group. The highlighted cells mark the cases in which more than 1% instances timed out.

benchmark	bit-width	method	error analysis						error checking	
			average-case				worst-case			
			ED	ED^2	EQ	HD	ED	HD	ED	EQ
adder	8	bdd	3.8 e^{5}	4.4 e^{5}	1.5 e^{5}	1.3 e^{5}	3.4 e^{5}	4.8 e^{5}	3.8 e^{5}	9.8 e^{4}
		sat	–	–	1.0 e^{7}	–	2.9 e^{6}	8.9 e^{5}	9.2 e^{5}	3.4 e^{5}
		sim	7.4 e^{6}	7.4 e^{6}	7.4 e^{6}	7.4 e^{6}	7.4 e^{6}	7.4 e^{6}	7.4 e^{6}	7.4 e^{6}
	12	bdd	1.3 e^{6}	1.7 e^{6}	3.3 e^{5}	2.3 e^{5}	1.0 e^{6}	2.5 e^{6}	1.3 e^{6}	1.7 e^{5}
		sat	–	–	8.8 e^{7}	–	5.8 e^{6}	1.7 e^{6}	1.8 e^{6}	5.6 e^{5}
		sim	2.4 e^{9}	2.4 e^{9}	2.4 e^{9}	2.4 e^{9}	2.4 e^{9}	2.4 e^{9}	2.4 e^{9}	2.4 e^{9}
	16	bdd	3.5 e^{6}	5.2 e^{6}	6.4 e^{5}	3.2 e^{5}	2.7 e^{6}	9.9 e^{6}	4.0 e^{6}	2.4 e^{5}
		sat	–	–	9.7 e^{8}	–	1.0 e^{7}	3.2 e^{6}	2.6 e^{6}	7.9 e^{5}
		sim	7.7 e^{11}	7.7 e^{11}	7.7 e^{11}	7.7 e^{11}	7.7 e^{11}	7.7 e^{11}	7.7 e^{11}	7.7 e^{11}
	32	bdd	2.1 e^{8}	3.6 e^{8}	1.3 e^{7}	8.9 e^{5}	1.3 e^{8}	9.3 e^{8}	3.1 e^{8}	5.9 e^{5}
		sat	–	–	5.5 e^{10}	–	3.3 e^{7}	1.6 e^{7}	1.6 e^{7}	1.8 e^{6}
multiplier	8	bdd	8.8 e^{7}	1.3 e^{8}	4.4 e^{7}	4.3 e^{7}	9.1 e^{7}	1.1 e^{8}	9.9 e^{7}	3.7 e^{7}
		sat	–	–	6.9 e^{10}	–	1.6 e^{10}	8.8 e^{7}	1.1 e^{7}	3.4 e^{6}
		sim	1.2 e^{7}	1.2 e^{7}	1.2 e^{7}	1.2 e^{7}	1.2 e^{7}	1.2 e^{7}	1.2 e^{7}	1.2 e^{7}
	12	bdd	1.2 e^{10}	3.4 e^{10}	6.0 e^{9}	5.6 e^{9}	1.2 e^{10}	2.7 e^{10}	1.6 e^{10}	4.0 e^{9}
		sat	–	–	–	–	7.3 e^{8}	1.9 e^{9}	1.0 e^{7}	9.2 e^{6}
		sim	4.2 e^{9}	4.2 e^{9}	4.2 e^{9}	4.2 e^{9}	4.2 e^{9}	4.2 e^{9}	4.2 e^{9}	4.2 e^{9}
	16	sat	–	–	–	–	9.9 e^{8}	4.7 e^{10}	2.1 e^{7}	1.7 e^{7}
		sim	1.4 e^{12}	1.4 e^{12}	1.4 e^{12}	1.4 e^{12}	1.4 e^{12}	1.4 e^{12}	1.4 e^{12}	1.4 e^{12}
	20	sat	–	–	–	–	1.7 e^{9}	1.5 e^{12}	2.9 e^{7}	2.7 e^{7}

solved with less effort compared to other metrics. This is noticeable especially on 32-bit adders where the difference in the number of instructions is in three orders of magnitude compared to the computation of the mean absolute error. The 32-bit approximate adder and 12-bit approximate multiplier, however, represent the largest instances that can be solved by BDDs. For adders, the computation of the maximum Hamming distance ended with a timeout for 5% of instances. It is impossible, in fact, to construct a BDD of a reasonable size for 16-bit multipliers. Given the experimental setup, all the runs ended with a timeout in this case.

Compared to the remaining methods, SAT solver-based methods scale best. They are capable of analyzing even 20-bit and more complex multipliers. The e_{wce} error analysis successfully finished for 81% of 12-bit, 75% of 16-bit and 75% of 20-bit instances. The worst-case checking was successful for 97% of 12-bit, 99.1% of 16-bit and 99.4% of 20-bit multipliers. The combinational equivalence checking (see the last column of Table 4) was successful in all cases. The #SAT solver (see the column 'EQ' in the error analysis part of Table 4, the row 'sat') performs surprisingly well. It is applicable even to 32-bit adders, but the results are obtained only for 39% of instances typically having the lower error (see Figure 8 for a more detailed analysis). In the remaining cases, the timeout occurred. As is evidently seen in Table 4, the computational requirements of #SAT solver are significantly higher compared to the BDDs. The number of instructions issued by the #SAT solver is about four orders of magnitude higher than that of the BDDs when we consider 32-bit adders.

TABLE 5. The average CPU time given in seconds required to perform the error analysis. We report the median value calculated from all experiments in the group. The highlighted cells mark the cases in which more than 1% instances timed out.

benchmark	bit-width	method	error analysis						error checking	
			average-case				worst-case			
			ED	ED^2	EQ	HD	ED	HD	ED	EQ
adder	8	bdd	0.001	0.001	0.001	0.000	0.001	0.001	0.001	0.000
		sat	–	–	0.005	–	0.001	0.000	0.000	0.000
		sim	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
	12	bdd	0.002	0.002	0.001	0.001	0.002	0.002	0.001	0.000
		sat	–	–	0.026	–	0.002	0.001	0.001	0.000
		sim	0.359	0.359	0.359	0.359	0.359	0.359	0.359	0.359
	16	bdd	0.003	0.004	0.001	0.001	0.003	0.008	0.003	0.000
		sat	–	–	0.273	–	0.003	0.001	0.001	0.000
		sim	107	107	107	107	107	107	107	107
	32	bdd	0.198	0.290	0.010	0.001	0.099	1	0.264	0.001
		sat	–	–	17	–	0.009	0.004	0.005	0.000
multiplier	8	bdd	0.068	0.090	0.034	0.033	0.073	0.087	0.083	0.028
		sat	–	–	28	–	5	0.026	0.003	0.001
		sim	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002
	12	bdd	15	37	7	7	15	36	19	4
		sat	–	–	–	–	0.224	0.600	0.002	0.002
		sim	0.646	0.646	0.646	0.646	0.646	0.646	0.646	0.646
	16	sat	–	–	–	–	0.307	16	0.005	0.004
		sim	198	198	198	198	198	198	198	198
	20	sat	–	–	–	–	0.541	528	0.007	0.007

Reporting solely the number of executed instructions may not be sufficient in general. There can be two algorithms with the same number of instructions, yet having a completely different execution time. It depends whether a particular code is a memory or computationally intensive. Hence, Table 5 reports the average CPU time recorded for each group of experiments to show how the number of instructions corresponds with the time of execution.

Only a few milliseconds are required on average to analyze 8-bit, 12-bit and 16-bit adders using BDDs. Less than three hundred milliseconds are required on average to analyze 32-bit adders. For multipliers, less than one hundred milliseconds are required to determine the error parameters. However, the time grows by two orders of magnitude when increasing the bit-width from eight to twelve bits. The SAT-solver based worst-case analysis performs well, even for 20-bit multipliers, but it is is computationally very intensive as will be discussed in Section V-E.

D. PERFORMANCE VARIATION

It is no surprise that the execution time depends on the number of primary inputs. In some cases, however, there is a huge variance even across instances of the same bit width. This suggests that the time required to perform the analysis also depends on properties of the individual instances.

Figure 7 shows the average CPU time for instances arranged into groups according to the worst-case error. The worst-case error was chosen as the indicator of circuit analysis complexity because there is a high degree of correlation

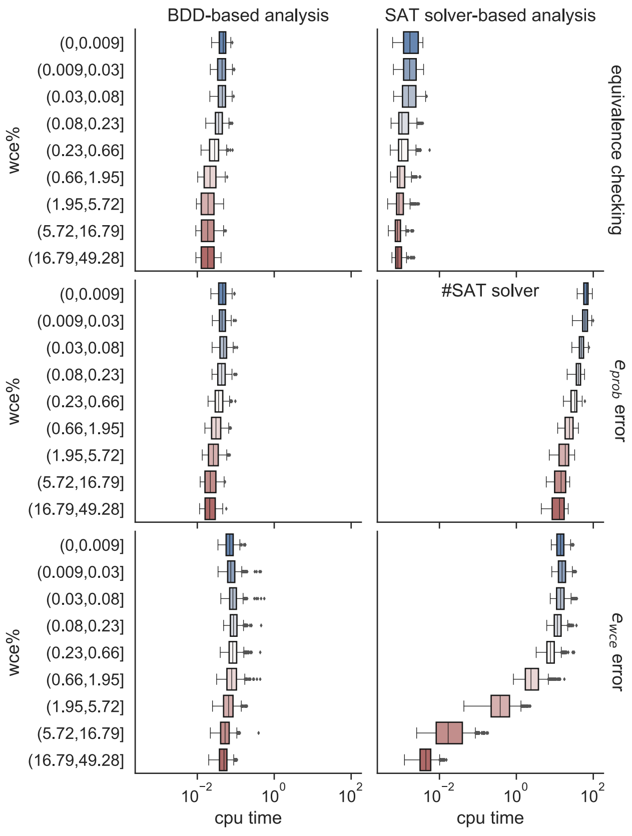


FIGURE 7. The CPU time of the error analysis shown as a function of the worst-case error of analyzed 8-bit approximate multipliers. Note that the log-scale is employed on the X-axis.

between this metric and the number of instructions required to perform the analysis (Pearson correlation coefficient is 0.93 for 8-bit multipliers and 0.94 for 32-bit adders analyzed by means of Algorithm 6). Similar dependence is also observed for the CPU time (Pearson correlation coefficient is 0.93 for 8-bit multipliers and 0.79 for 32-bit adders). The boxplots are calculated from data acquired during the analysis of 8-bit multipliers. This benchmark was chosen because it represents a non-trivial problem where we have results for all methods and instances (i.e. no timeout occurred). Ten points taken uniformly in the log domain were used to establish the intervals for the worst-case error. The results for simulation are intentionally omitted because the number of instructions is nearly constant. The runtime depends only on the number of gates of an analyzed circuit rather than the internal structure. The first row of Figure 7 contains data for common equivalence checking to provide a baseline for comparison.

The experimental results show that the CPU time significantly depends on properties of the investigated circuits. This observation is also valid for the number of instructions. In the case of multipliers, the computational complexity of the analysis decreases with increasing the worst-case error independently of the chosen formal approach. The difference is substantial, especially for the SAT-solved based worst-case error analysis. The approximate multipliers exhibiting e_{wce} in the range between 16% and 50% can

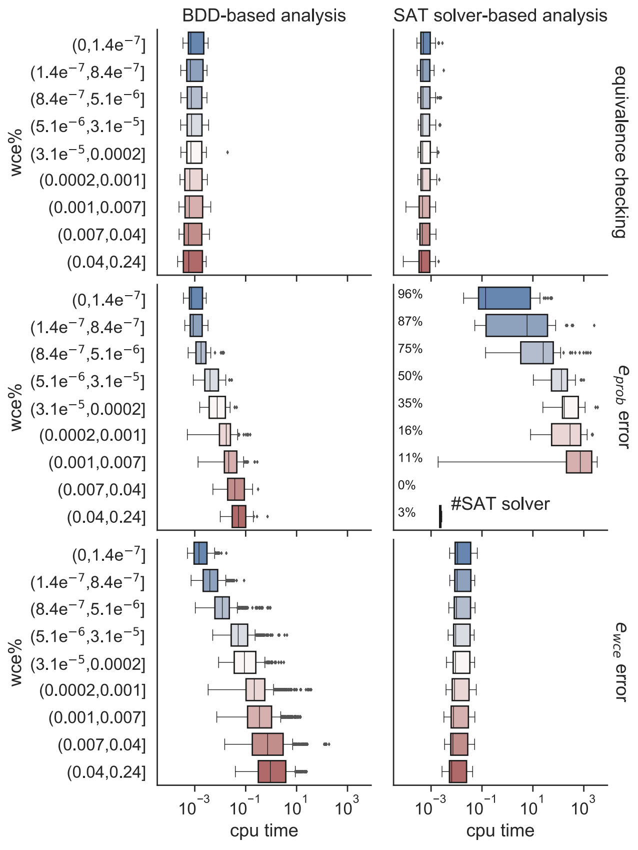


FIGURE 8. The CPU time of the error analysis depicted as a function of the normalized worst-case error of analyzed 32-bit approximate adders. The percentages shown on the left side of the boxplots represent the relative number of successfully analyzed instances included in the boxplot and indicate that timeout occurred.

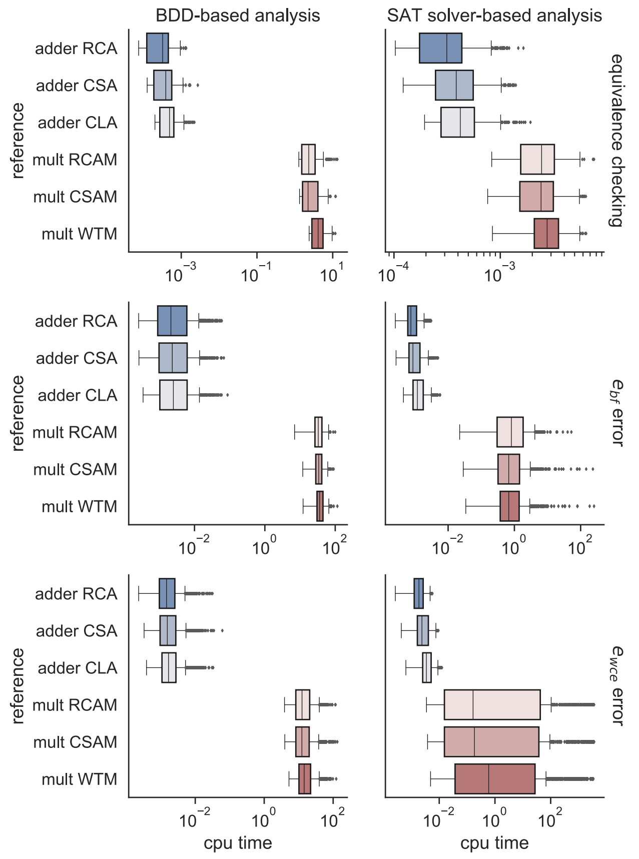


FIGURE 9. The CPU time of the error analysis and its dependency on reference circuit used to analyze 12-bit adders and multipliers.

be analyzed approximately three thousand times faster than those having a low error. A similar dependency is also observable for more complex instances (i.e. 12-bit and 16-bit multipliers). The BDDs exhibit relative stable performance when applied to the worst-case error analysis.

Interestingly, exactly the opposite trend is observable in the case of adders (see Figure 8). Here the CPU time increases when the worst-case error increases. Determining the error probability using BDDs can take up to ten times more time for adders having a larger error compared to those having a low error. Combinational equivalence checking and SAT-based worst-case error analysis exhibit a relatively stable performance (1.5 times vs 1.1 times higher number of instructions for higher errors). We intentionally chose 32-bit instances to illustrate how #SAT solver behaves on more complex instances. It is evident that this method exhibits the same behavior. In this case, the CPU time fluctuates within two orders of magnitude depending on a particular instance. The large variation in the range (0.001%, 0.007%] is caused by few approximate circuits exhibiting 100% error probability. Those instances are trivial for SAT solving and could be analyzed in a few milliseconds. The same situation happened also for circuits in the range (0.04%, 0.24%]. Apart from those trivial instances, no result was obtained within the given amount of time for circuits having the worst-case error greater than 0.0067%. The explanation of this behavior is as follows. The high worst-case error is usually connected with a high error probability. Unfortunately, the analysis of approximate circuits exhibiting high error probability is nontrivial for #SAT probably because there are many input assignments that have to be considered during the analysis of the miter depicted in Figure 4a. The #SAT solver finished successfully for 30% of the instances with an error probability higher than 95%. Note that the large variation in the case of BDD-based computation of e_{wce} is due to the presence of several algorithms.

The last analysis shown in Figure 9 investigates the dependence between the CPU time and a reference circuit used in the approximation miter. Three different implementations of accurate adders and multipliers have been considered in our experiments, particularly the Ripple Carry Adder (RCA), Carry Save Adder (CSA) and sophisticated Carry Look Ahead adder (CLA). The most visible difference is in the case of combinational equivalence checking. Surprisingly, the reference has a substantial impact on equivalence checking, regardless of the used formal method. The RCA ensures the best performance despite it has a long XOR chain which may negatively affect the performance of formal approaches. Considering the multipliers, we evaluated CSA-based Array

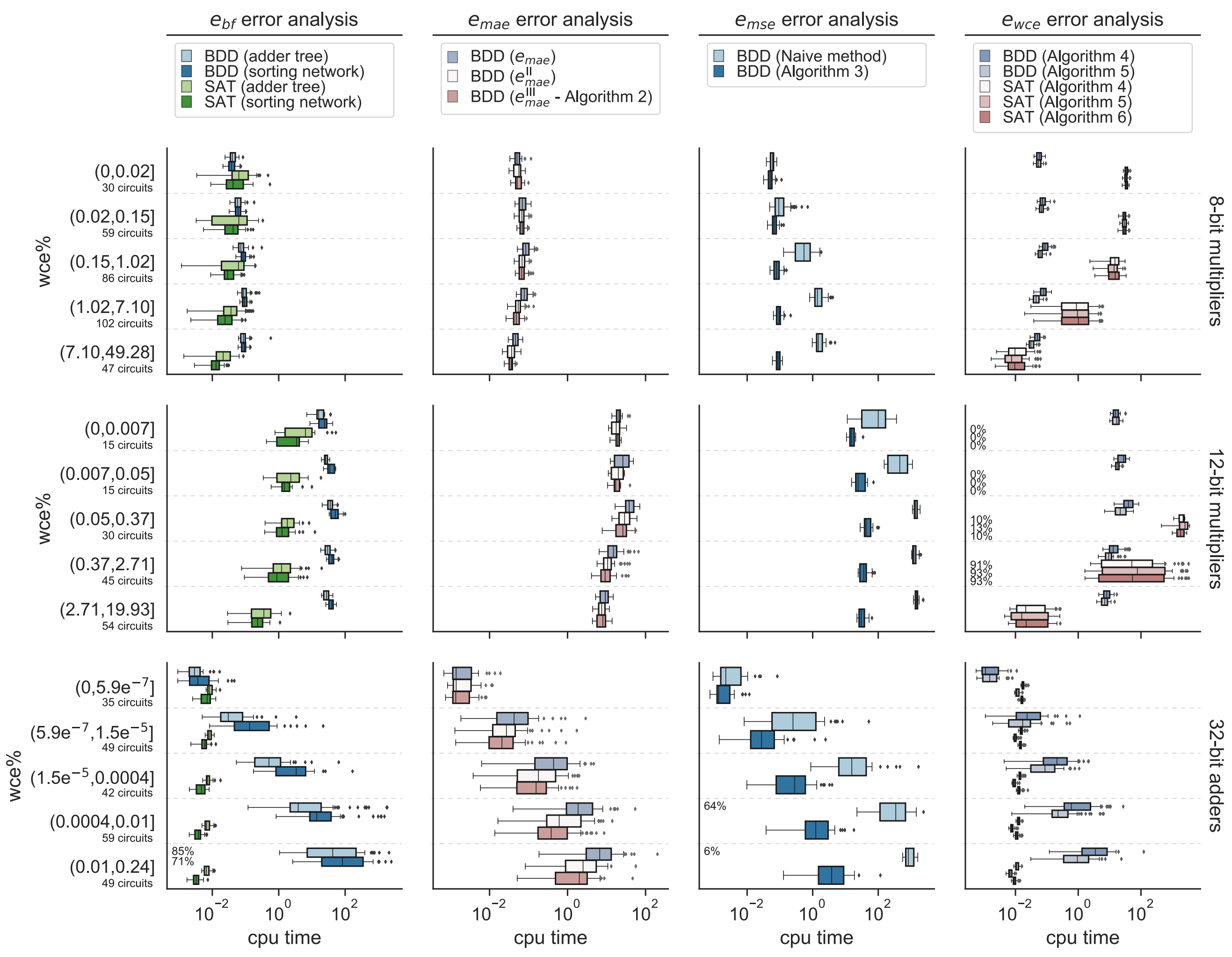


FIGURE 10. **Evaluation of the proposed error analysis algorithms on 8-bit multipliers, 12-bit multipliers, and 32-bit adders. Each column compares the results of different approaches computing the same error metric. The percentages shown on the left side of the boxplots represent the relative number of successfully analyzed instances included in the boxplot and indicate that timeout occurred. The number of circuit instances within a certain wce% range is given below each range.**

Multiplier (CSAM), RCA-based Array Multiplier (RCAM) and Wallace-Tree Multiplier (WTM). The WTM performs worst. The remaining two architectures provide quite similar results.

E. EVALUATION OF THE AVAILABLE AND PROPOSED ALGORITHMS

In this section, we evaluate the performance of the algorithms proposed in this paper. Since the performance of the formal methods varies depending on the parameters of the analyzed approximate circuits and their type, we report the results separately for adders and multipliers arranged into groups according to their worst-case error. This form helps us to provide meaningful insight into the real computational requirements. Six points taken uniformly in the log domain were used to establish the intervals for the worst-case error.

a: BIT-FLIP ERROR

The results for the worst-case Hamming distance computation are summarized in the first column of Figure 10. Four implementations are evaluated. The first and second implementation is based on the approximation miter with the sorting network as proposed in Section IV-D. This miter is then represented and analyzed using BDDs or the SAT solver. Both implementations follow Algorithm 1; however, only the SAT solver further exploits the satisfiable assignment. Bitonic sorting network was used in both cases. The second and third implementations are based on the approximation miter consisting of a tree of adders computing the number of active bits. The output is then analyzed using either BDD-based or SAT-based algorithm for determining the maximum output value (Algorithm 4).

We discuss the implementations based on BDDs first. Surprisingly the BDD-based analysis based on the adder tree is more efficient compared to the one with the sorting network.

This result is quite surprising since our intuition was that the miter based on the sorting network can be represented more efficiently compared to the tree of adders. On the other hand, it is necessary to realize that the performance of BDDs depends on the chosen variable ordering. We chose the ordering optimal for representing adders which could potentially cause a problem, especially if the resulting structure is extremely different from the adder. Unfortunately, determining the optimal ordering is a non-trivial NP hard problem.

The SAT-solver based analysis performs better and in most cases even substantially (see the boxplots for 12-bit multipliers and 32-bit adders). It also scales better as it can be applied to a 16-bit and more complex multipliers, where the BDD-based implementations did not provide any result (see Table 4). Compared to the BDDs, the SAT solvers clearly profit from the approximation miter with a sorting network and the analysis based on this miter (i.e. the proposed Algorithm 1) represents the most powerful approach. The analysis was successful for all instances included in Figure 10. Both BDD-based implementations ended with a timeout for some 32-bit adders having the worst-case error in the range (0.01%, 0.24%].

b: MEAN ABSOLUTE ERROR

The second column of Figure 10 compares the CPU time of three methods for calculating the mean absolute error, in particular the method available in the literature and two alternative versions proposed in this paper. The actual implementation follows exactly the procedure derived in Section IV-E, in particular Eq. 18, 20 and 21. The analysis is performed by means of BDDs because it relies on SAT counting.

The experimental evaluations confirmed that Algorithm 2 implementing the procedure derived in Eq. 21 is the most efficient approach for determining the mean absolute error. It provides a speedup between 1 to 5.3 on average compared to the original method. The second proposed method performs also well, but it has slightly worse parameters. It achieves a speedup between 1 to 2.9 on average.

The analysis was successful for all circuit instances included in the comparison. In the worst case, 71 seconds was required to analyze the most difficult approximate circuits. The computational complexity of Algorithm 2 is lower compared to the bit-flip error analysis. This is noticeable especially on 32-bit adders.

c: MEAN SQUARED ERROR

Two algorithms for determining the mean square error were implemented. A naïve approach based on a miter with a multiplier corresponding with Figure 5c, and the proposed alternative method according to Algorithm 3. Note that the implementation of the algorithm is the trickiest one since the algorithm involves summing many large values multiplied by a relatively high power of two. It is thus necessary to use high precision arithmetic [2] to avoid overflows. Both methods are evaluated and included in Figure 10.

The naïve method does not scale well. The computational complexity increases with the increasing worst-case error. Around 29% of instances of 32-bit approximate adders ended with the timeout. Those instances have the wce larger than 0.0004%. The computation mostly failed during the construction of the approximation miter. This is somewhat expected as the miter consists of a 33-bit multiplier. If the difference between the approximate and reference circuit leads to a complex BDD structure, the multiplier causes the exponential explosion considering the number of BDD nodes. As a consequence of that, the naïve method ended with a timeout in some cases.

On the other hand, the proposed method was successful in all cases. The achieved speedup of the proposed method is remarkable. It varies between 1.2 (8-bit multipliers and 32-bit adders with the lowest worst-case error) and 317 (32-bit adders with higher worst-case error). In the worst case, 113 seconds was required to analyze the most difficult approximate circuit. The computational complexity is comparable to the mean absolute error computation.

d: WORST-CASE ARITHMETIC ERROR

Three methods for the worst-case error analysis were implemented and evaluated: Algorithm 4, its optimized variant denoted as Algorithm 5, and Algorithm 6 based on a binary search and WCEGT operation. The first two algorithms were implemented as BDD-based as well as SAT solver-based tools. The latter algorithm is not suitable for BDDs and BDDs were not considered in this case. Regarding SAT solver-based implementations, incremental SAT solving and learning of some bits from the received SAT assignments is employed. The masking is ensured via assumptions. Algorithm 6 is implemented as follows. Firstly, CNF of the approximation miter is created and submitted to the SAT solver. Then, CNF corresponding with a threshold circuit generated for a particular threshold is appended to the original CNF in each iteration. After solving this CNF, the output of this threshold circuit is blocked by setting a corresponding assumption and the computation continues with the next iteration. The results are summarized in the last column of Figure 10.

Let us discuss the complexity of error analysis of the multipliers at first. Although the SAT solver-based methods scale better and can be applied to complex multipliers where the BDDs completely failed, the time of execution of SAT solver-based analysis is mostly worse compared to the BDDs, especially on 8-bit and 12-bit multipliers. A reasonable speedup is observable only for approximate circuits having a large error. All three SAT solver-based methods behave similarly considering the number of successfully solved instances. The implementations based on Algorithm 5 and Algorithm 6 represent the most powerful methods. Both successfully analyzed 99 out of 159 12-bit approximate multipliers.

[2] GNU Multi-Precision Arithmetic (GMP) library was used in this work.

This is quite surprising because a completely different approach is used in the case of Algorithm 6. Implementation based on Algorithm 4, however, provides nearly the same results because 98 instances were successfully analyzed. Regarding the CPU time, there is no significant difference among these three implementations when applied to the approximate multipliers. Unfortunately, the CPU time heavily depends on the actual worst-case error. As the worst-case error increases, the CPU time increases as well. Only a few milliseconds are required to analyze easily solvable instances exhibiting large error. Considering the analysis of 12-bit approximate multipliers, more than 2,000 seconds are needed to analyze the instances having the error in the range (0.05%,0.37%]. No result was obtained within the given amount of time for multipliers exhibiting an error below 0.05%.

A completely opposite situation is observable in the case of approximate adders. Algorithm 5 substantially outperforms the remaining two. In addition, the SAT solver-based implementations achieve significantly better execution times compared to the BDDs and offer a relative robust approach with a stable performance. The execution time is between 4 to 25 ms. On the other hand, the BDD-based algorithms exhibit a huge variance ranging from microseconds to more than 110 seconds. Figure 10 reports the results only for 32-bit adders, but the benefits of Algorithm 5 are observable also for 8-bit, 12-bit, and 16-bit instances. For 32-bit adders, the speedup of the BDD-based implementation varies between 1.2 and 3. The SAT solver-based implementation achieved the speedup between 1.5 and 2. The experimental results clearly demonstrate the superiority of Algorithm 5 to Algorithm 4 despite of the fact that the difference between these two approaches is less visible in the case of multipliers. Considering this result, it can be concluded that it is, in general, advantageous to use the miter providing the signed output even though it is necessary to increase the complexity of the algorithm (the signed output implied an extra additional operation in Algorithm 5).

Figure 11 helps us to better understand the performance of the SAT-solver based worst-case error analysis of multipliers. It shows the computational requirements of the WCEGT procedure (i.e. worst-case error checking) for different thresholds applied to all 324 8-bit instances. Five thresholds linearly sampled in the log space and one additional point ($\mathcal{T} = 3333$) are considered. The results are presented for 8-bit approximate multipliers included in the benchmark dataset, but the conclusions are also valid for other bit widths. The 8-bit instances are chosen because we have error parameters for every multiplier and no timeout occurred. The worst-case error checking is extremely fast (only a few milliseconds are required) but only if the actual worst-case error (denoted as wce) is higher than a given threshold $\mathcal{T}$. If this condition is violated, the CPU time may increase by several orders of magnitude. Surprisingly, the difference between the worst case and the best case CPU time increases with the decreasing of threshold $\mathcal{T}$. Determining whether the worst-case error

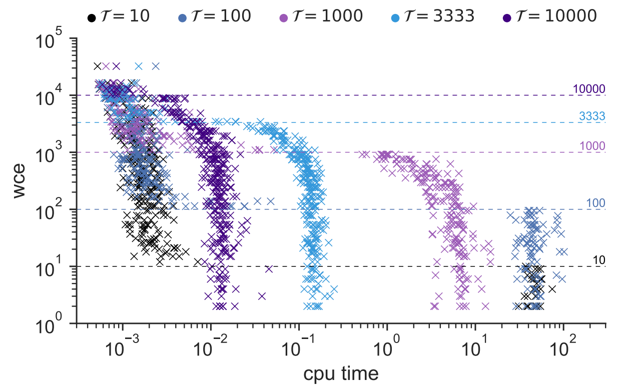


FIGURE 11. **The computational requirements of the worst-case error checking (i.e. the WCEGT procedure proving that $e_{wce} > \mathcal{T}$) of 8-bit approximate multipliers. 'Mult RCAM' used as a reference circuit.**

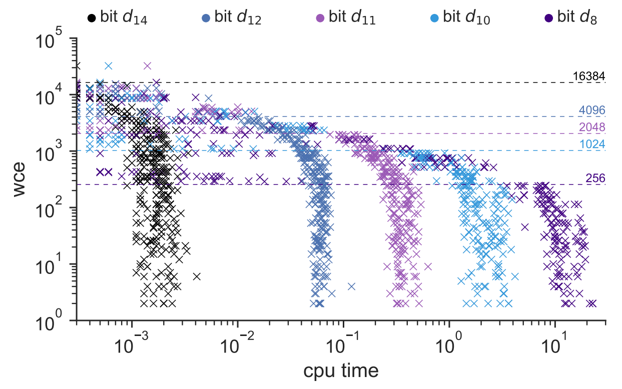


FIGURE 12. **CPU time needed to determine the value of some bits of e_{wce} for 8-bit approximate multipliers analyzed using Algorithm 4. 'Mult RCAM' used as reference circuit. e_{wce} is encoded using 16 bits denoted $d_0, \ldots, d_{15}$.**

is greater than $\mathcal{T} = 100$ represents the most difficult case. Up to 100 seconds are required to analyse circuit instances having a wce lower than 100. This dependence explains the performance of Algorithm 6 in Figure 10. To determine the worst-case error of 8-bit approximate multipliers, 16 calls of the WCEGT procedure are required at most. The runtime of WCEGT increases as the threshold approaches the real wce and reaches a peak around 100. Hence, the total runtime increases with a decreasing wce and is highest for the lowest wce.

In spite of the fact that Algorithm 4 and 5 are based on a different principle (compared to Algorithm 6), they behave similarly considering the CPU time. In Figure 12 we plotted the time needed to determine the value of some bits of the wce to understand this feature. We present the results for Algorithm 4, but the observations are valid also for its improved version. The algorithm determines the worst-case error bit per bit starting with the most significant bit (denoted as bit d_{15}). Determining the value of d_{15} is extremely fast (only a few milliseconds are required). The CPU time then increases with every next bit and determining the value of d_8 represents the most difficult case (more than 10 seconds are required). For the next bits, i.e. $d_7, \ldots, d_0$, the CPU time gradually decreases. As a consequence, the total runtime of the worst-case error analysis of 8-bit multipliers is dominated by the CPU time required to determine the value of bits d_7, d_8, d_9. This part represents 95% of the whole runtime on average. Similar to Figure 11, the checking is fast only if the actual worst-case error is higher than the threshold corresponding with the weight of the checked bit (2^i for d_i).

VI. CONCLUSION

It is a well-known fact that the equivalence checking of multipliers is an extremely hard problem; however, there is no study that investigates the complexity of determining error metrics for approximate circuits, especially from the practical point of view. We presented a comprehensive evaluation of various methods across a large set of various approximate circuits. We discussed the scalability of the methods as well as their stability (i.e. dependence on some properties of the analyzed circuits). This is the first study that experimentally demonstrated that the performance of the methods heavily depends on parameters of analyzed circuits independently of whether we choose BDDs or SAT solvers to perform the analysis.

We determined that the worst-case error represents an important error parameter. The worst-case error seems to be a good indicator of the complexity of error analysis because the CPU time typically increases by increasing the worst-case error. In addition, the correlation analysis revealed that many other error parameters relevant for practice, such as the mean absolute error or mean relative error, correlate with the worst-case error.

Considering the analysis of approximate circuits, the experimental results demonstrated that BDDs and SAT solvers are orthogonal methods. Each technique has its own strengths and performs well only in some tasks and for some type of circuits. The BDDs naturally compute all assignments in parallel and are thus suitable for average-case error analysis (computation of e_{mae}, e_{mse}). In contrast, SAT solvers are efficient regarding memory consumption, but only give a single satisfiable assignment in each call. This limitation leads to iterative approaches that should be based on a relative low number of iterations (such as e_{bf}, e_{wce}, e_{wcre}) to guarantee reasonable scalability.

As a result of the study, we identified the best algorithms and tools for each error metric. For small circuits having up to 16 inputs (e.g. two 8-bit operands), performance-optimized exhaustive circuit simulation represents the method of the first choice. The simulation can be applied even to more complex problem instances (circuits with up to 32 inputs), but the formal approaches such as BDDs or SAT-solvers typically scale better. In these cases, Algorithm 1 implemented using a SAT solver is the best option for bit-flip error analysis (e_{bf}). The analysis of the mean absolute error (e_{mae}) can be performed efficiently using BDDs and the proposed Algorithm 2. For mean squared error (e_{mse}), Algorithm 3 suitable for BDDs was designed. The worst-case error analysis (e_{wce}) can be done efficiently by the proposed Algorithm 5. Depending on the structure of the analysed approximate circuits, however, a BDD or a SAT solver needs to be chosen in this case. Finally, a variant of Algorithm 6 seems to be a good option for the relative worst-case error analysis (e_{wcre}).

Considering the worst-case error analysis, the SAT-based implementations provide very good results, but many challenges remain. It is possible to analyze even instances that are usually considered to be hard (e.g. 12-bit and larger multipliers), but no results were obtained in the predefined amount of time (3,600 seconds) for instances exhibiting a lower error. It has been shown that the computational complexity dramatically increases with a decreasing error, especially on multipliers. Unfortunately, the multiplier is one of the key arithmetic circuits that is widely used in many applications, especially in digital signal processing and multimedia processing. Hence, there is currently a clear need to come up with a more powerful approach to the problem of evaluating the quality of complex approximate digital circuits. A combination of the circuit simulator and a SAT-solver seems to be a promising approach which helps us to further improve the performance of the analysis.

REFERENCES

[1] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 62-1–62-33, Mar. 2016.
[2] Q. Xu, M. Todd, and S. K. Nam, "Approximate computing: A survey," *IEEE Design Test*, vol. 33, no. 1, pp. 8–22, Feb. 2016.
[3] S. Reda and M. Shafique, Eds., *Approximate Circuits*. Cham, Switzerland: Springer, 2019.
[4] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits," in *Proc. Design, Autom. Test Eur. (DATE)*. San Jose, CA, USA: EDA Consortium, 2013, pp. 1–6.
[5] K. Nepal, Y. Li, R. I. Bahar, and S. Reda, "Abacus: A technique for automated behavioral synthesis of approximate computing circuits," in *Proc. Conf. Design, Autom. Test Eur. (DATE)*. San Jose, CA, USA: EDA Consortium, 2014, pp. 1–6.
[6] L. Sekanina, Z. Vasicek, and V. Mrazek, "Automated search-based functional approximation for digital circuits," in *Approximate Circuits*. Cham, Switzerland: Springer, Dec. 2018, pp. 175–203.
[7] C. Liu, J. Han, and F. Lombardi, "An analytical framework for evaluating the error characteristics of approximate adders," *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1268–1281, May 2015.
[8] 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 Trans. Comput.*, vol. 68, no. 1, pp. 21–38, Jan. 2019.
[9] S. Mazahir, M. K. Ayub, O. Hasan, and M. Shafique, "Probabilistic error analysis of approximate adders and multipliers," in *Approximate Circuits*. Cham, Switzerland: Springer, Dec. 2018, pp. 99–120.
[10] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan, "MACACO: Modeling and analysis of circuits for approximate computing," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2011, pp. 667–673.
[11] D. Sengupta, J. Hu, and S. S. Sapatnekar, "Error analysis and optimization in approximate arithmetic circuits," in *Approximate Circuits*. Cham, Switzerland: Springer, Dec. 2018, pp. 225–246.
[12] V. Mrazek, Z. Vasicek, L. Sekanina, H. Jiang, and J. Han, "Scalable construction of approximate multipliers with formally guaranteed worst case error," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 11, pp. 2572–2576, Nov. 2018.
[13] M. Soeken, D. Große, A. Chandrasekharan, and R. Drechsler, "BDD minimization for approximate computing," in *Proc. 1st Asia South Pacific Design Automat. Conf. (ASP-DAC)*, Jan. 2016, pp. 474–479.
[14] Z. Vasicek, V. Mrazek, and L. S. Brno, "Towards low power approximate DCT architecture for HEVC standard," in *Proc. Design, Autom. Test Eur. (DATE)*, Mar. 2017, pp. 1576–1581.

[15] Z. Vasicek, "Relaxed equivalence checking: A new challenge in logic synthesis," in *Proc. IEEE 20th Int. Symp. Design Diag. Electron. Circuits Syst. (DDECS)*, Apr. 2017, pp. 1–6.

[16] M. Ceska, J. Matyas, V. Mrazek, L. Sekanina, Z. Vasicek, and T. Vojnar, "Approximating complex arithmetic circuits with formal error guarantees: 32-bit multipliers accomplished," in *Proc. 36th IEEE/ACM Int. Conf. Comput. Aided Design*, Nov. 2017, pp. 416–423.

[17] Y. Wu and W. Qian, "ALFANS: Multi-level approximate logic synthesis framework by approximate node simplification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, to be published.

[18] S. Froehlich, D. Große, and R. Drechsler, "Approximate hardware generation using symbolic computer algebra employing grobner basis," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 889–892.

[19] S. Froehlich, D. Große, and R. Drechsler, "One method-all error-metrics: A three-stage approach for error-metric evaluation in approximate computing," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 284–287.

[20] P. Kulkarni, P. Gupta, and M. D. Ercegovac, "Trading accuracy for power in a multiplier architecture," *J. Low Power Electron.*, vol. 7, no. 4, pp. 490–501, 2011.

[21] J. Miao, K. He, A. Gerstlauer, and M. Orshansky, "Modeling and synthesis of quality-energy optimal approximate adders," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*. New York, NY, USA, 2012, pp. 728–735.

[22] V. Mrazek and Z. Vasicek, "Automatic design of arbitrary-size approximate sorting networks with error guarantee," in *Proc. 26th Int. Workshop Power Timing Modeling, Optim. Simulation*, 2016, pp. 221–228.

[23] Z. Vasicek and V. Mrazek, "Trading between quality and non-functional properties of median filter in embedded systems," *Genetic Program. Evolvable Mach.*, vol. 18, no. 1, pp. 45–82, 2017.

[24] A. Chandrasekharan, M. Soeken, D. Große, and R. Drechsler, "Precise error determination of approximated components in sequential circuits with model checking," in *Proc. 53rd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2016, pp. 1–6.

[25] M. Traiola, A. Virazel, P. Girard, M. Barbarcschi, and A. Bosio, "Investigation of mean-error metrics for testing approximate integrated circuits," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFT)*, Oct. 2018, pp. 1–6.

[26] S. Mazahir, O. Hasan, R. Hafiz, M. Shafique, and J. Henkel, "Probabilistic error modeling for approximate adders," *IEEE Trans. Comput.*, vol. 66, no. 3, pp. 515–530, Mar. 2017.

[27] H. Jiang, C. Liu, N. Maheshwari, F. Lombardi, and J. Han, "A comparative evaluation of approximate multipliers," in *Proc. IEEE/ACM Int. Symp. Nanoscale Archit. (NANOARCH)*, Beijing, China, Jul. 2016, pp. 191–196.

[28] D. S. Khudia and B. Zamirai, "Rumba: An online quality management system for approximate computing," in *Proc. ISCA*, 2015, pp. 554–566.

[29] M. Traiola, A. Savino, M. Barbareschi, S. D. Carlo, and A. Bosio, "Predicting the impact of functional approximation: From component-to application-level," in *Proc. IEEE 24th Int. Symp. On-Line Test. Robust Syst. Design (IOLTS)*, Jul. 2018, pp. 61–64.

[30] A. Biere and W. Kunz, "Sat and atpg: Boolean engines for formal hardware verification," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Nov. 2002, pp. 782–785.

[31] A. Chandrasekharan, S. Eggersglüß, D. Große, and R. Drechsler, "Approximation-aware testing for approximate circuits," in *Proc. 23rd Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2018, pp. 239–244.

[32] T.-H. Chen, A. Alaghi, and J. P. Hayes, "Behavior of stochastic circuits under severe error conditions," *Inf. Technol.*, vol. 56, no. 4, pp. 182–191, 2014.

[33] W. T. J. Chan, A. B. Kahng, S. Kang, R. Kumar, and J. Sartori, "Statistical analysis and modeling for error composition in approximate computation circuits," in *Proc. 31st IEEE Int. Conf. Comput. Design (ICCD)*, Oct. 2013, pp. 47–53.

[34] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina, "Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, 2017, pp. 258–261.

[35] J. Liang, J. Han, and F. Lombardi, "New metrics for the reliability of approximate and probabilistic adders," *IEEE Trans. Comput.*, vol. 62, no. 9, pp. 1760–1771, Sep. 2013.

[36] A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability*. Amsterdam, The Netherlands: IOS Press, 2009.

[37] J. Marques-Silva, "Practical applications of Boolean satisfiability," in *Proc. Workshop Discrete Event Syst. (WODES)*. Piscataway, NJ, USA: IEEE Press, May 2008, pp. 74–80.

[38] M. W. Krentel, "The complexity of optimization problems," *J. Comput. Syst. Sci.*, vol. 36, no. 3, pp. 490–509, 1988.

[39] A. Petkovska, A. Mishchenko, M. Soeken, G. De Micheli, R. Brayton, and P. Ienne, "Fast generation of lexicographic satisfiable assignments: Enabling canonicity in sat-based applications," in *Proc. ACM 35th Int. Conf. Comput.-Aided Design (ICCAD)*, New York, NY, USA, 2016, pp. 4-1–4-8.

[40] C. P. Gomes, A. Sabharwal, and B. Selman, "Model counting," in *Handbook of Satisfiability*, A. Biere, M. Heule, H. Van Maaren, and T. Walsh, Eds. Amsterdam, The Netherlands: IOS Press, 2009, ch. 20, pp. 266–290.

[41] A. Qureshi and O. Hasan, "Formal probabilistic analysis of low latency approximate adders," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 1, pp. 177–189, Jan. 2019.

[42] R. Drechsler and B. Becker, *Binary Decision Diagrams: Theory and Implementation*. Springer, 2013.

[43] R. E. Bryant, "On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication," *IEEE Trans. Comput.*, vol. 40, no. 2, pp. 205–213, Feb. 1991.

[44] R. Ebendt, G. Fey, and R. Drechsler, *Advanced BDD Optimization*. Springer, 2000.

[45] Z. Vasicek and L. Sekanina, "Evolutionary design of complex approximate combinational circuits," *Genetic Program. Evolvable Mach.*, vol. 17, no. 2, pp. 169–192, 2016.

[46] A. Nadel and V. Ryvchin, "Efficient SAT solving under assumptions," in *Theory and Applications of Satisfiability Testing—SAT*, A. Cimatti and R. Sebastiani, Eds. Berlin, Germany: Springer, 2012, pp. 242–255.

[47] Y. Voronenko and M. Püschel, "Multiplierless multiple constant multiplication," *ACM Trans. Algorithms*, vol. 3, no. 2, May 2007, Art. no. 11.

[48] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proc. CAV*. Berlin, Germany: Springer, 2010.

[49] J. Lind-Nielsen and H. Cohen. *BuDDy—A Binary Decision Diagram Package*. Accessed: Aug. 12, 2019. [Online]. Available: https://sourceforge.net/projects/buddy/

[50] M. Thurley, "sharpSAT—Counting models with advanced component caching and implicit BCP," in *Proc. 9th Int. Conf. Theory Appl. Satisfiability Test. (SAT)*, Berlin, Germany: Springer-Verlag, 2006, pp. 424–429.

[51] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," in *Tools for High Performance Computing*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Berlin, Germany: Springer, 2010, pp. 157–173.

ZDENEK VASICEK is currently an Associate Professor and a member of the Evolvable Hardware Group, Faculty of Information Technology, Brno University of Technology, Czech Republic. He is interested in the optimization and synthesis of digital circuits and applications of evolutionary approaches and formal techniques in areas related to this problem. He has (co) authored over 50 articles on (non) evolutionary design and optimization of common and approximate digital circuits at renowned international conferences, such as ICCAD, DATE, DAC, GECCO, and EuroGP and in international journals, such as the IEEE VLSI, the IEEE TEC, and GPEM. His research interests include logic synthesis, evolvable hardware, genetic programming, and approximate computing.