One Method - All Error-Metrics: A Three-Stage Approach for Error-Metric Evaluation in Approximate Computing

Saman Froehlich Daniel Große Rolf Drechsler

Cyber-Physical Systems, DFKI GmbH and Group of Computer Architecture, University of Bremen, Germany
saman.froehlich@dfki.de {grosse,drechsle}@cs.uni-bremen.de

***Abstract*—Approximate Computing (AC) is a design paradigm that makes use of the error tolerance inherited by many applications. The goal of AC is to trade off accuracy for performance in terms of computation time, energy consumption and/or hardware complexity.**

In the field of circuit design for AC, error-metrics are used to express the degree of approximation. Evaluating these error-metrics is a key challenge. Several approaches exist, however, to this day not all relevant metrics can be evaluated with formal methods. Recently, Symbolic Computer Algebra (SCA) has been used to evaluate error-metrics during approximate hardware generation. In this paper, we generalize the idea to use SCA and propose a methodology which is suitable for formal evaluation of *all* established error-metrics. This approach can be divided into three stages: 1) Determine the remainder of the AC circuit wrt. the specification using SCA, 2) build an Algebraic Decision Diagram (ADD) to represent the remainder and 3) evaluate each error-metric by a tailored ADD traversal algorithm. In the experiments, we apply our algorithms to a large and well-known benchmark set.

I. Introduction

Approximate Computing (AC) is a design paradigm which makes use of the error tolerance inherited by many applications, such as machine learning, media processing and data mining. The goal of AC is to trade off accuracy for performance in terms of computation time, energy consumption and/or hardware complexity [1], [2].

When designing AC circuits *error-metrics* are of major importance. An error-metric evaluates the difference between the approximation and the specification in terms of a given metric. The initial methods for error-metric evaluation of AC circuits were based on simulation and statistical analysis (see for instance [1]). However, since exhaustive simulation is not feasible for larger circuits, the user has to trust these approaches in the sense that "sufficiently representative" scenarios have been considered. For this reason formal approaches have been investigated, since their major advantage is to enable guarantees wrt. the given metric. In the last years several approaches for formal error-metric evaluation have been proposed (e.g. [3], [4], [5], [6]).

Recently, in [7] an alternative has been presented which is based on *Symbolic Computer Algebra* (SCA) – the recent theoretical and practical enhancements of SCA allows to verify the correctness of large arithmetic circuits. In verification, the essentials of SCA are to model the gates as polynomials, and then to divide the specification polynomial of the circuit stepwise by these gate-polynomials. If the remainder of this division becomes zero, the circuit is an exact implementation of the specification polynomial, otherwise the remainder describes the error. This principle has been exploited in [7] to evaluate two error-metrics during hardware generation.

This work was supported in part by the German Research Foundation (DFG) within the project MANIAC (DR 287/29-1) and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

In this paper we generalize the idea to use SCA and propose a **three-stage approach for formal evaluation of *all* established error-metrics in AC**. The three stages are: 1.) Determine the remainder of the AC circuit wrt. the specification using SCA (Gröbner Reduction). 2.) Build an Algebraic Decision Diagram (ADD) to represent the remainder. 3.) Evaluate each error-metric by a tailored ADD traversal algorithm. Besides the generic three-stage approach, the major contributions of our work are the error-metric specific ADD traversal algorithms. We succeeded to develop ADD algorithms which allow to analyze *all* established error-metrics. This includes in particular the *Worst-Case-Relative Error* and *Average-Case-Relative Error* for which no other formal evaluation techniques exist.

II. Related Work

The authors of [3] have presented a BDD-based algorithm for error-metric evaluation. The proposed approach is limited to the worst-case error, the average-case error and the error rate. Further, their approach does not allow incorporating input probabilities.

In [8] several approximate architectures evaluated for different error-metrics are presented. However, these architectures are evaluated using a non-formal statistical approach.

[4] presents a miter based method for the evaluation of sequential circuits, which is based on the miters introduced in [1]. This approach is limited to the worst-case error, the average-case error and the error rate. [5] has extended this approach by simplifying the evaluation process and could successfully use it to generate approximate 32bit multipliers. However, again this approach is limited to a few error-metrics and can not incorporate input probabilities. Recently, in [9] the methodology of [5] has been integrated into Berkeley-ABC [10].

The authors of [7] were the first to use SCA for error-metric evaluation. They presented methods for the evaluation of the worst-case error and the mean-squared error only, while we present algorithms for all relevant error-metrics. Their approach is relatively slower compared to ADDs.

Table I gives an overview of the existing techniques for error-metric evaluation and compares them to ours. The first column denotes the formal methods. The 2nd-7th columns denote which error-metric can be evaluated by each technique respectively. It can be seen, that we are the first to present an

TABLE I
OVERVIEW OF FORMAL TECHNIQUES FOR ERROR-METRIC EVALUATION

Algorithm	wc-error	wcr-error	ac-error	acr-error	ms-error	error-rate	bf-error
BDD-Based [3]	x		x			x	
Miter-Based [4], [1], [5]	x					x	x
SCA-Based [7]	x				x		
Proposed Approach	**x**	**x**	**x**	**x**	**x**	**x**	**x**

approach which is applicable for all other error-metrics, giving a closed technique for error-metric evaluation.

III. Preliminaries

Due to page limitation we cannot give a detailed introduction to the application of SCA to circuits. If the reader is not familiar with the application of SCA in this context, we refer to [11], [12], [13]. We give a brief introduction to ADDs ind Section III-A and introduce the relevant error-metrics in Section III-B.

A. Algebraic Decision Diagrams

Algebraic Decision Diagrams (ADDs) [14] are word-level decision diagrams which can be used to represent pseudo-Boolean functions $f : \mathbb{B}^n \rightarrow \mathbb{R}$ and are based on the Shannon decomposition $f = x_i f_{x_i} + \bar{x}_i f_{\bar{x}_i}$ $(1 \leq i \leq n)$.

B. Error-Metrics

Over the past years several metrics have been used in approximate computing. A complete list of the most popular ones can be found in [8]. We give a short recapitulation with $f(x)$ being the output of the exact implementation and $\hat{f}(x)$ being the output of the approximate circuit:

- One of the most popular metrics is the *Worst-Case Error* (wc-error). It describes the maximum error the approximation may give.
$$wce(f, \hat{f}) = \max_x \{|f(x) - \hat{f}(x)|\}. \tag{1}$$
- Closely related to the wc-error is the *Worst-Case-Relative Error* (wcr-error). It is a measure for the maximum error in relation to the correct output.
$$wcre(f, \hat{f}) = \max_x \{\frac{|f(x) - \hat{f}(x)|}{\max(1, |f(x)|)}\}. \tag{2}$$
- The *Average-Case Error* (ac-error) describes the average error induced by approximation.
$$ace(f, \hat{f}) = \frac{\sum\limits_x |f(x) - \hat{f}(x)|}{2^m}. \tag{3}$$
- The *Average-Case-Relative Error* (acr-error) is related to the ac-error and describes the average error relative to the amplitude of the correct value. Thus it allows for larger errors at larger amplitudes.
$$acre(f, \hat{f}) = \frac{\frac{\sum\limits_x |f(x) - \hat{f}(x)|}{\max(1, |f(x)|)}}{2^m}. \tag{4}$$
- The *Mean-Squared Error* (ms-error) describes the average squared error induced by approximation. This error-metric is relevant because it is inversely related to the PSNR, which is a common measure for the quality of images.
$$mse(f, \hat{f}) = \frac{\sum\limits_x (f(x) - \hat{f}(x))^2}{2^m}. \tag{5}$$
- The *Error Rate* (er) describes the probability that the output of the approximation deviates from the true result. It is defined as
$$er(f, \hat{f}) = \frac{\sum\limits_{x \in \mathbb{B}^m} f(x) \neq \hat{f}(x)}{2^m} \tag{6}$$
- The *Bit-Flip Error* (bf-error) is related to the maximum Hamming Distance between the approximation and the true result. It is defined as
$$bfe(f, \hat{f}) = \max_{x \in \mathbb{B}^m} \sum_{x=0}^{n-1} |f_i(x) \neq \hat{f}_i(x)| \tag{7}$$

All of the above presented metrics are relevant for different applications (sometimes in combination) and refer to different aspects of the degree of approximation. While for some applications the maximum error-magnitude (wc-error) might be limited, the error-rate might be of interest for other applications.

IV. ADD Traversal Algorithms

In order to evaluate the error-metrics given a word-level formulation of the desired behavior and a gate-level description of the circuit, we propose a method which is divided into three stages: 1.) Determine the remainder of the AC circuit wrt. the specification using SCA (Gröbner Reduction) 2.) Build an ADD to represent the remainder 3.) Evaluate each error-metric by a tailored ADD traversal algorithm. In this section we describe the third stage: The tailored ADD traversal algorithms. Since most of the proposed ADD traversal algorithms for error-metric evaluation are based on an algorithm for *Minterm* (MT) counting, we introduce this algorithm first and afterwards describe the changes which are to be made to this algorithm for each error-metric respectively. Algorithm 1 depicts the Pseudocode taken from the implementation of $Cudd\_CountMinterm$ in [15].

Given a function $f : \mathbb{B}^m \rightarrow \mathbb{R}$ represented as ADD, the algorithm consists of two functions: A non recursive function $countMT$ (Line 1) and a recursive helper function $countMTRecur$ (Line 8). The function $countMinterms$ first calculates the maximum number of minterms in Line 4 and consecutively calls the recursive helper function in Line 5 to calculate the actual number of minterms. $countMintermsRecursive$ takes the current node N, the maximum number of minterms max and a hash table $hashTable$ containing the calculated results for

TABLE II
COMPUTATION TIMES OF ERROR-METRICS FOR APPROXIMATE CIRCUITS

Name	Gröbner Reduction [ms]	ADD Creation [ms]	wc-error [ms]	wcr-error [ms]	ac-error [ms]	acr-error [ms]	ms-error [ms]	error rate [ms]	bf-error [ms]
8-bit Adders									
$ACA\_I\_N8\_Q5$	2.465	12.502	0.044	13.278 (11.653)	0.061	12.591 (10.966)	0.020	0.017	0.016
$ACA\_II\_N8\_Q4$	1.236	8.057	0.022	12.629 (11.087)	0.056	13.108 (11.566)	0.020	0.017	0.014
$GDA\_St\_N8\_M4\_P2$	0.701	5.565	0.016	8.630 (7.598)	0.046	9.694 (8.662)	0.015	0.013	0.010
$GDA\_St\_N8\_M4\_P4$	1.588	9.539	0.021	11.814 (10.235)	0.041	8.919 (7.340)	0.011	0.010	0.009
$GDA\_St\_N8\_M8\_P1$	0.520	5.669	0.018	26.098 (24.687)	0.072	30.026 (28.615)	0.046	0.037	0.036
$GDA\_St\_N8\_M8\_P2$	1.152	6.533	0.053	12.376 (10.795)	0.078	12.616 (11.035)	0.041	0.035	0.029
$GDA\_St\_N8\_M8\_P3$	1.487	6.232	0.043	10.786 (9.370)	0.046	12.905 (11.489)	0.022	0.017	0.014
$GDA\_St\_N8\_M8\_P4$	2.701	8.947	0.049	11.737 (10.025)	0.049	10.821 (9.109)	0.019	0.017	0.015
$GDA\_St\_N8\_M8\_P5$	4.193	13.405	0.049	11.497 (9.733)	0.049	9.440 (7.676)	0.018	0.016	0.014
$GeAr\_N8\_R1\_P1$	0.287	4.578	0.028	15.485 (14.706)	0.056	23.0178 (22.238)	0.033	0.027	0.028
$GeAr\_N8\_R1\_P2$	0.977	7.127	0.053	13.099 (11.454)	0.078	12.986 (11.341)	0.050	0.044	0.038
$GeAr\_N8\_R1\_P3$	1511	7.518	0.048	12.597 (11.004)	0.067	12.952 (11.359)	0.022	0.018	0.015
$GeAr\_N8\_R1\_P4$	2.019	8.559	0.038	10.213 (8.799)	0.048	11.250 (9.836)	0.017	0.014	0.013
$GeAr\_N8\_R1\_P5$	2.872	13.575	0.050	11.704 (9.948)	0.038	11.114 (9.358)	0.018	0.016	0.014
$GeAr\_N8\_R2\_P2$	1.141	6.792	0.024	12.192 (10.554)	0.053	13.201 (11.563)	0.022	0.017	0.014
$GeAr\_N8\_R2\_P4$	2.148	9.304	0.024	11.078 (9.538)	0.040	10.808 (9.268)	0.011	0.010	0.009
16-bit Adders									
$ACA\_I\_N16\_Q4$	4.810	64.710	0.161	43204.9 (26252.6)	0.454	43704.1 (26751.8)	0.547	0.315	0.256
$ACA\_II\_N16\_Q4$	3.312	23.229	0.091	44633.7 (27752.3)	0.195	45476.9 (28595.5)	0.301	0.130	0.083
$ACA\_II\_N16\_Q8$	23.691	88.428	0.012	15243.3 (30.9)	0.053	1544.0 (50.6)	0.026	0.017	0.013
$ETAII\_N16\_Q4$	3.354	24.703	0.099	44775.4 (27818.9)	0.213	45225.9 (28269.4)	0.350	0.124	0.081
$ETAII\_N16\_Q8$	25.782	90.342	0.013	15360.3 (30.9)	0.079	15379.4 (45.0)	0.023	0.019	0.013
$GDA\_St\_N16\_M4\_P4$	38.652	104.141	0.017	22379.2 (6121.1)	0.092	22292.1 (6034.0)	0.026	.0214	0.013
$GDA\_St\_N16\_M4\_P8$	1063.900	3307.99	0.017	19588.0 (4.0)	0.076	19588.6 (4.6)	0.025	0.020	0.012
$GeAr\_N16\_R2\_P4$	11.760	66.208	0.046	24265.0 (9979.6)	0.143	24361.2 (10075.8)	0.091	0.053	0.037
$GeAr\_N16\_R4\_P4$	33.327	104.834	0.013	15419.7 (30.9)	0.054	15438.7 (49.9)	0.026	0.018	0.013
$GeAr\_N16\_R4\_P8$	591.195	2.908.780	0.017	17261.3 (18.0)	0.088	17262.4 (19.1)	0.023	0.020	0.011
$GeAr\_N16\_R6\_P4$	104.718	626.022	0.011	12763.4 (20.2)	0.075	12771.5 (28.3)	0.017	0.012	0.009
32-bit Adders									
$ACA\_I\_N32\_Q8$	258	5861	3.416	timeout	9.372	timeout	6.916	5.482	5.213
$ACA\_II\_N32\_Q16$	133332	1241450	0.166	timeout	2916.94	timeout	7.700	6.776	6.772

Algorithm 1 countMinterms

```
1: function COUNTMT(f)
2:   // f is the function to be approximated represented as ADD
3:   var hashTable
4:   max = pow(2, f.GetNumberOfVariables, hashTable)
5:   return COUNTMTRECUR(f.rootNode, max, hashTable)
6: end function
7:
8: function COUNTMTRECUR(N, max, hashTable)
9:   if isTerminal(N) then
10:    if isZero(N) then
11:      return 0
12:    end if
13:    return max
14:  end if
15:  if hashTable.find(N) then
16:    return hashTable.result(N)
17:  end if
18:  resultT = COUNTMTRECUR(N.T, max, hashTable)
19:  resultE = COUNTMTRECUR(N.E, max, hashTable)
20:  result = 0.5 · resultT + 0.5 · resultE
21:  hashTable.insert(N, result)
22:  return result
23: end function
```

each node as inputs. If a terminal node is reached, the function either returns 0 if it is the 0 terminal or max otherwise (Lines 9-14). Otherwise, it checks if a result for the current node has already been calculated (Line 15) and returns the corresponding value if that is the case. If this is not the case, the function is called recursively for each child node and the result is the sum of the results of the child nodes multiplied by 0.5 (Line 20). Finally, the calculated result is stored in the hash table and returned.

We propose to represent the remainder of the Gröbner Reduction as an ADD and calculate the error-metrics using the following, tailored ADD traversal algorithms:

a) Worst-Case Error: The wc-error is equal to the largest absolute value the remainder can attain (see Eq. 1). In order to calculate the wc-error using the ADD representation it is sufficient to extract the terminal with the largest absolute value.

b) Worst-Case-Relative Error: For the calculation of the wcr-error (defined in Eq. 2), we build the ADDs for the remainder as well as $\max(1, f(x))$ and use the *apply* algorithm to calculate the ADD representation for the division. Subsequently, we use the same algorithm as for the wc-error.

c) Average-Case Error: To calculate the ac-error (Eq. 3), we count the paths to each possible result. Based on the ADD representation we modify Line 13 in Algorithm 1. Instead of returning max, we return $max \cdot |value(N)|$, where $value(N)$ is the value of the current terminal node. Finally, we divide the result by 2^m.

d) Average-Case-Relative Error: In order to calculate the acr-error (see Eq. 4) based on the ADD representation, we build the ADDs for the remainder as well as $\max(1, f(x))$ and use the $apply$ algorithm to calculate the ADD representation for the division. Finally, we use the same algorithm as for the ac-error.

e) Mean-Squared Error: For the calculation of the ms-error as defined in Eq. 5 based on the ADD representation, we modify Line 13 in Algorithm 1. Instead of returning max, we return $max \ \cdot value(N)^2$, where $value(N)$ is the value of the current terminal node. Finally, we divide the result by 2^m.

f) Error Rate: In order to calculate the error rate (see Eq. 6), we calculate the number of minterms and divide it by 2^m. To increase the efficiency of the algorithm, we set the values of all non-zero terminal nodes to 1 and reduce the diagram, effectively creating a BDD.

g) Bit-Flip Error: For the calculation of the bf-error (Eq. 7), we find the terminal node of the ADD representation with the highest number of 1s in the binary representation of its value. The result is the number of 1s of the binary representation of the value represented by this node.

One of the benefits of the SCA-based approach for error-metric calculation is that the algorithm for the ms-error provided in [7] allows to evaluate the error-metric wrt. a given distribution of input probabilities. We incorporate this into our algorithms by changing Line 20 in Algorithm 1: Instead of multiplying $resultT$ and $resultE$ by 0.5, we multiply $resultT$ by p and $resultE$ by $(1-p)$, where p is the probability of the variable represented by the node N to evaluate to *true* (input probability). Using the same modifications as described in the paragraphs above, we can evaluate the ac-error, the acr-error, the ms-error and the error rate with respect to input probabilities.

V. Experimental Evaluation

All experiments have been carried out on an Intel® Xeon® CPU E5-2630 v3 @ 2.40GHz with 64GB memory running Linux (Fedora release 22). We have used CUDD 3.0.0 [15] as a library for ADDs.

Both the Gröbner Reduction and building the ADD has to be performed only once, even if the same circuit has to be evaluated for different error-metrics. For this reason we give the computation time for each step individually in our results.

We apply our algorithms to the well-known KIT-Benchmarkset [16]. The results are shown in Table II. The first column denotes the name of the circuit. The second column gives the computation time of the Gröbner Reduction. The third column denotes how long it took to build the ADD using CUDD with activated sifting. Finally, columns 4-11 give the computation times of each error-metric respectively, given the ADD representation. For the wcr-error and the acr-error (columns 5 and 7), we show the total computation time including building the ADD for the original circuit and the time for evaluating the metric itself (i.e. dividing the ADDs and applying the algorithms) in brackets.

It can be seen that evaluating the wcr- and the acr-error-metrics is in general a lot harder than evaluating the other five error metrics. This is caused by the divisions of the ADD representations. However for small circuits (8-bit and 16-bit adders), the computation time is still reasonable (below 60 seconds in total for all cases). For circuits with more than 32 inputs, building the ADD representation of the correct behavior can be challenging. We had a timeout for the 32-bit adders after 4 hours (which have 64 inputs effectively). However, the computation of the ADD representation of the correct behavior can in general be done offline for relevant circuits (such as adders and multipliers) to reduce computation time of these metrics.

For the other error-metrics, the computation of the Gröber Reduction and building the ADD representation of the remainder are the most time-consuming parts of the evaluation. Since these tasks have to performed only once no matter how many error-metrics are to be evaluated, our approach is especially well suited for applications where more than one error-metric is relevant.

VI. Conclusion

In this paper, we are the first to present a single formal method for the evaluation of all relevant error-metrics in approximate computing. Despite existing methods, our three-stage approach is applicable to all error-metrics.

We have used our approach to evaluate a large set of benchmarks. In our experiments, we have shown that performing the Gröbner Reduction and computing the ADD representation of the remainder are the most time-consuming parts of the evaluation. This makes our approach specially effective for applications where more than one error-metric has to be evaluated, since these tasks need to be performed only once.

References

[1] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan, “MACACO: modeling and analysis of circuits for approximate computing,” in *ICCAD*, Nov. 2011, pp. 667–673.

[2] S. Froehlich, D. Große, and R. Drechsler, “Towards reversed approximate hardware design,” in *DSD*, 2018, pp. 665–671.

[3] M. Soeken, D. Große, A. Chandrasekharan, and R. Drechsler, “BDD minimization for approximate computing,” in *ASP-DAC*, 2016, pp. 474–479.

[4] A. Chandrasekharan, M. Soeken, D. Große, and R. Drechsler, “Precise error determination of approximated components in sequential circuits with model checking,” in *DAC*, 2016.

[5] M. Češka, J. Matyaš, V. Mrazek, L. Sekanina, Z. Vasicek, and T. Vojnar, “Approximating complex arithmetic circuits with formal error guarantees: 32-bit multipliers accomplished,” in *ICCAD*, 2017, pp. 416–423.

[6] A. Chandrasekharan, D. Große, and R. Drechsler, *Design Automation Techniques for Approximation Circuits*. Springer, 2018.

[7] S. Froehlich, D. Große, and R. Drechsler, “Approximate hardware generation using symbolic computer algebra employing Gröbner basis,” in *DATE*, 2018.

[8] 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 *DATE*, 2017.

[9] M. Češka, J. Matyáš, V. Mrázek, L. Sekanina, Z. Vašíček, and T. Vojnar, “Adac: Automated design of approximate circuits,” in *CAV*, 2018, pp. 1–9. [Online]. Available: http://www.fit.vutbr.cz/research/view_pub.php.cz?id=11731

[10] A. Mischenko, M. Case, R. Brayton, and S. Jang, “Scalable and scalably-verifiable sequential synthesis,” in *ICCAD*, 2008.

[11] F. Farahmandi and B. Alizadeh, “Groebner basis based formal verification of large arithmetic circuits using gaussian elimination and cone-based polynomial extraction,” *MICPRO*, vol. 39, no. 2, pp. 83 – 96, 2015.

[12] A. S. Ahmed, D. Große, U. Kühne, M. Soeken, and R. Drechsler, “Formal verification of integer multipliers by combining Gröbner basis with logic reduction,” in *DATE*, 2016, pp. 1048–1053.

[13] A. Mahzoon, D. Große, and R. Drechsler, “PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers,” in *ICCAD*, 2018, pp. 129:1–129:8.

[14] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi, “Algebric decision diagrams and their applications,” *Formal Methods in System Design: An International Journal*, vol. 10, no. 2, pp. 171–206, Apr 1997.

[15] F. Somenzi, “CUDD: CU Decision Diagram package-release 3.0.0,” University of Colorado at Boulder, 2015.

[16] Chair for Embedded Systems - Karlsruhe Institute of Technology, “Gear - approxadderlib.” [Online]. Available: http://ces.itec.kit.edu/GeAR.php