A fast BDD Minimization Framework for Approximate Computing

Andreas Wendler Oliver Keszocze

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
andreas.wendler@fau.de oliver.keszoecze@fau.de

***Abstract*—Approximate Computing is a design paradigm that trades off computational accuracy for gains in non-functional aspects such as reduced area, increased computation speed, or power reduction. Computing the error of the approximated design is an essential step to determine its quality. The computation time for determining the error can become very large, effectively rendering the entire logic approximation procedure infeasible.**

As a remedy, we present methods to accelerate the computation of error metric computations by (a) exploiting structural information and (b) computing estimates of the metrics for multi-output Boolean functions represented as BDDs. We further present a novel greedy, bucket-based BDD minimization framework employing the newly proposed error metric computations to produce Pareto-optimal solutions with respect to BDD size and multiple error metrics. The applicability of the proposed minimization framework is demonstrated by an experimental evaluation. We can report considerable speedups while, at the same time, creating high-quality approximated BDDs.

***Index Terms*—Approximate Computing, Design Space Exploration, Multi-Objective Optimization, BDD Minimization, Logic Minimization, Error Metrics**

I. Introduction

Many applications in the domain of digital signal processing, such as image processing, do not require computations to be exact (see, e.g. [1]). This is due to the fact that the human perception itself is not perfect. In other situations, it might be difficult or even impossible to define what an exact result would be or the customer is willing to accept non-perfect results [2].

Approximate Computing [3, 4] is a design paradigm that exploits this fact and trades off computational accuracy for gains in non-functional aspects such as reduced area, increased computation speed, or power reduction. There are two main approaches to introduce changes to the design (see, e.g. [5]) in order to achieve gains in, e.g., computation speed: (a) physical changes such as voltage over-scaling or over-clocking (b) changes in the functionality of the design. In this paper, we follow the second approach.

Let f be a Boolean function describing some functionality that is implemented correctly, i.e. does not produce erroneous outputs. By $\hat{f}$ we then denote a function that approximates the original function f, i.e. $f \approx \hat{f}$ with some notion of similarity.

It turns out that determining the similarity between f and $\hat{f}$ is surprisingly difficult. This greatly hinders design methods, such as evolutionary algorithms, that rely on evaluating the similarity repeatedly. There is an obvious need for methods that determine the similarity of a design f and its approximation $\hat{f}$ quickly or, at least, compute an estimate of the similarity in a reasonable amount of time.

This work has been partially funded by the Emerging Talents Initiative of the FAU under the number ETI 2019/1_Tech_03

II. Related Work

A. Error Metric Computation

The notion of similarity is captured by *error metrics* that range from simply counting how often a design yields incorrect results to computing the mean squared error (see [6] for a good overview).

Making use of BDDs for the computation of error metrics is a common approach. In [7], miter circuits are represented as BDDs in order to quickly determine the error rate between the golden implementation and the approximation. The authors of [8] introduce multiple BDD-based algorithms to compute even more complex error metrics. In [9], the authors make use of Algebraic Decision Diagrams and Gröbner Bases in a unified framework that allows the computation of all error metrics introduced in [6].

All approaches, however, have in common that they reach computational limits sooner or later. Studies on the complexity of the error metrics have shown that they reside in complexity classes #P and FP^{NP} [10], explaining the scalability issues. In the literature, researchers try to cope with this complexity by using Monte Carlo Simulations, see e.g. [11].

B. Approximate Logic Synthesis

A common approach for approximate logic synthesis is to perform a design space exploration using Cartesian Genetic Programming in order to compute Pareto sets of approximated circuits. In these works, either only a simple error metric is used [7] or great efforts are made to speed up the error metric computation by using a formulation specifically tailored to work well with SAT solvers [12]. The authors of [6] are capable of computing a high-quality Pareto front by restraining themselves to 8-bit adders and multipliers.

In [13], an evolutionary algorithm working directly on the BDD representation of designs is proposed. It automatically adapts its preference for either the BDD size or the error metric value in order to ensure that no good solution is lost due to optimizing too much for a small error.

The authors of [14] choose to use cubes to represent the designs being approximated. The cubes can be used both for generating approximation candidates as well as for directly computing the error rate.

III. Preliminaries

A. Notation and Conventions

In this paper, all functions will be of type $f : \mathbb{B}^n \to \mathbb{B}^m$. The m individual output functions are denoted as f_i. The interpretation of $f(x)$ as a natural number with the usual binary encoding is denoted by $\mathrm{val}(f(x))$. While all the results presented in this paper extend to signed numbers, we omit the details for simplicity. A function's ON-set is denoted by $ON(f)$. As a shorthand, we define the absolute difference function of f and its approximation $\hat{f}$ as $d(x) := |\mathrm{val}(f(x)) - \mathrm{val}(\hat{f}(x))|$.

We make use of Binary Decision Diagrams (BDDs) [15] to represent Boolean functions. For simplicity, a collection of BDDs representing a multi-output functions will also be referred to as "a BDD".

B. Approximation Operations on BDDs

There are multiple methods of approximating functions given as BDDs in order to reduce the BDD's size. In this paper, we restrict ourselves to the approximation operations presented in [8] and exemplarily introduce the *round-down/up* operations.

Definition 1 (Round-down/up): The two operations *round-down* and *round-up*, $\lfloor f \rfloor_{x_i}$ and $\lceil f \rceil_{x_i}$ take each node at level x_i and below and replace the child node with the smaller ON-set with the 0- or 1-terminal, respectively. If both children have the same ON-set, the low-child is chosen.

Example 1: Consider the function f represented by a BDD, shown in Fig. 1(a). Applying the round-down operation at level x_2, the BDD is reduced by two nodes. The affected edges are highlighted by the $\times$ symbol. The resulting BDD is shown in Fig. 1(b).

C. Error Metrics

There are many different error metrics for assessing the quality of an approximated function $\hat{f}$, each measuring a different aspect of the introduced error. Mrazek et al. give a good overview over commonly used metrics in [6][1]. While our framework is capable of computing all error metrics from [6], due to page limitations, only a selection will be presented in this paper.

The *error rate* er counts how often the approximation and the original function differ without taking into account any semantic meaning of the output. The *average-case error* ace computes the average absolute difference in the output of f and $\hat{f}$ while the *average-case relative error* acre relates the difference to the correct value of f:

$$\mathrm{er}(f,\hat{f}) := \frac{1}{2^n} \sum_{x\in\mathbb{B}^n} \Big(f(x) \neq \hat{f}(x)\Big) \tag{1}$$

$$\mathrm{ace}(f,\hat{f}) := \frac{1}{2^n} \sum_{x\in\mathbb{B}^n} |\mathrm{val}(f(x)) - \mathrm{val}(\hat{f}(x))| \tag{2}$$

$$\mathrm{acre}(f,\hat{f}) := \frac{1}{2^n} \sum_{x\in\mathbb{B}^n} \frac{|\mathrm{val}(f(x)) - \mathrm{val}(\hat{f}(x))|}{\max\{1, |\mathrm{val}(f(x))|\}} \tag{3}$$

[1]Note that we use a slightly different naming scheme for the error metrics.

TABLE I: Computation times for different steps of error metric computations in milliseconds.

Circuit	Op.	$d = \lvert f - \hat{f}\rvert$	$\mathrm{ace}(f,\hat{f})$	% of d
c432	$\lceil f \rceil_{x_{20}}$	87	92	95
c432	$\lceil f \rceil_{x_{30}}$	53	57	94
c499	$\lceil f \rceil_{x_{35}}$	2923	2962	> 99
c1355	$\lceil f \rceil_{x_{10}}$	669	749	89

Example 2: Consider again the functions f and $\hat{f}$ from Example 1. In Fig. 1(c), the corresponding truth table and the numerical values are shown. This allows to determine the following error metric values:

$$\mathrm{er}(f,\hat{f}) = 3/8, \quad \mathrm{ace}(f,\hat{f}) = 5/8, \quad \text{and } \mathrm{acre}(f,\hat{f}) = 1/3.$$

IV. Accelerating Error Metric Computations

A. Average-Case Error Computation

1) Avoiding Construction of the Characteristic Function: In [8], the authors construct the BDD for the characteristic function of d to compute the average-case error. It can easily be seen in [8, Table III] that switching to an algorithm constructing the characteristic function greatly increases the computation time. It turns out that this construction can be avoided entirely and ace can simply be computed by determining the size of m ON-sets instead:

$$\begin{aligned}\mathrm{ace}(f,\hat{f}) &= \frac{1}{2^n} \sum_{x\in\mathbb{B}^n} \mathrm{val}(d(x)) = \frac{1}{2^n} \sum_{x\in\mathbb{B}^n} \sum_{i=1}^{m} 2^{i-1} \cdot d_i(x) \\ &= \frac{1}{2^n} \sum_{i=1}^{m} 2^{i-1} \cdot \#\,\mathrm{ON}(d_i). \end{aligned} \tag{4}$$

Determining the ON-set can be done in linear time with respect to the nodes of the BDD representing d.

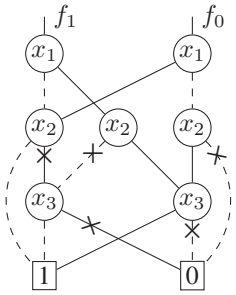
2) Exploiting Approximation Operation Properties: The time needed to construct d itself already makes up a considerable percentage of the total computation time in many cases. Table I contains a representative analysis of partial computation times for the ace computation showing this[2]. It is therefore desirable not to construct d at all. It turns out that this is possible if the inequality $\mathrm{val}(\hat{f}(x)) \leq \mathrm{val}(f(x))$ holds. In this case, one does not need to construct any new function but can directly compute the average-case error by re-suing the idea from the previous section as

$$\begin{aligned}\mathrm{ace}(f,\hat{f}) &= \frac{1}{2^n} \sum_{x\in\mathbb{B}^n} \mathrm{val}(f(x)) - \frac{1}{2^n} \sum_{x\in\mathbb{B}^n} \mathrm{val}(\hat{f}(x)) \\ &= \frac{1}{2^n} \sum_{i=1}^{m} 2^{i-1} \#ON(f_i) - \frac{1}{2^n} \sum_{i=1}^{m} 2^{i-1} \#ON(\hat{f}_i). \end{aligned}$$

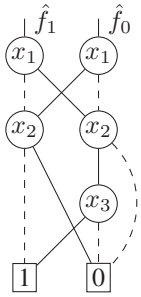
It turns out that the round-down operation from [8] can guarantee the even stronger subset property $\hat{f}_i(x) \leq f_i(x)$ for $1 \leq i \leq m$ (see also Fig. 1(c)). Using a full subtractor circuit for the computation of d, the individual output bits are given by $d_i = f_i \oplus \hat{f}_i \oplus c_i$ with the carry bits being computed as

$$c_{i+1} = (\hat{f}_i \wedge c_i) \vee (\overline{f_i} \wedge (\hat{f}_i \vee c_i)).$$

[2]See Section VI for details on the experimental setup.



(a) BDD for $f = (f_1, f_0)$



(b) BDD for $\hat{f} = \lfloor f \rfloor_{x_2}$

x	$f(x)$	$\hat{f}(x)$	$f \neq \hat{f}$	$\mathrm{val}(f(x))$	$\mathrm{val}(\hat{f}(x))$
000	10	10	0	2	2
001	10	10	0	2	2
010	10	00	1	2	0
011	01	01	0	1	1
100	11	01	1	3	1
101	01	01	0	1	1
110	01	00	1	1	0
111	10	10	0	2	2

(c) Truth Table and numerical values for f and $\hat{f}$

Fig. 1: Example of the application of the round-down operation.

By induction, it can easily be shown that all c_i are 0. Assume that $c_i = 0$ (which holds for the initial carry bit c_0), then

$$c_{i+1} = (\hat{f}_i \wedge 0) \vee (\overline{f_i} \wedge (\hat{f}_i \vee 0)) = \overline{f_i} \wedge \hat{f}_i \overset{\text{subset prop.}}{=} 0.$$

This allows a quick computation of error metrics such as ace, as the construction of d becomes trivial, i.e. $d_i = f_i \oplus \hat{f}_i$.

As the average-case error is symmetric, i.e. $\mathrm{ace}(f, g) = \mathrm{ace}(g, f)$, the cases $\mathrm{val}(f(x)) \leq \mathrm{val}(\hat{f}(x))$ and $f_i(x) \leq \hat{f}_i(x)$ can be handled trivially by swapping the arguments to ace. This makes the simplification applicable to the round-up operation.

As long as the round-up and round-down operations are not mixed in consecutive approximation steps, the proposed construction of d can be used. This will be exploited by the greedy, bucket-based BDD minimization algorithm presented in Section V.

B. Approximating the Average-Case Relative Error

Computing the average-case relative error is very expensive; in [9], the authors had to report timeouts for some of their benchmarks. Consequently, we propose an approximation method for the error metric, computing an upper and lower bound on acre. We further define the arithmetic mean of these bounds to be an approximation of acre, i.e. $\hat{\mathrm{acre}}(f, \hat{f}) := {}^{low+up}/_{2}$.

Our approach is shown in Algorithm 1. The idea is to iteratively assume that the i-th bit of f evaluates to 1 and the more significant bits evaluate to 0. The variable $mask$ represents this condition (see its creation in line 4). This mask is applied to the computed absolute difference to create an approximated difference $\hat{d}$ (line 5) and its average value is computed (line 6). We then compute the upper bound for the error of bit i by further assuming all less significant bits are 0 and dividing by 2^{i-1}. The lower bound is computed analogously. Note that in line 1 the simplifications discussed in Section IV-A2 are applied if possible and that in line 6 the idea from Section IV-A1 is re-used.

The difference between the correct value for acre and its approximation $\hat{\mathrm{acre}}$ has an upper bound of

$$|\mathrm{acre}(f, \hat{f}) - \hat{\mathrm{acre}}(f, \hat{f})| \leq \frac{up - low}{2}.$$

Algorithm 1: Determining the lower and upper bound of the acre error metric

input : Functions f and $\hat{f}$
output: Lower and upper bound on $\mathrm{acre}(f, \hat{f})$

1 $d \leftarrow |f - \hat{f}|$
2 $low, up \leftarrow 0$
3 **for** $i = 1$ *to* m **do**
4 $\quad mask \leftarrow f_i \wedge \left(\bigwedge_{j=i+1}^{m} \overline{f_j}\right)$ ▷ $mask = f_m$ *if* $i = m$
5 $\quad \hat{d} \leftarrow (d_1 \wedge mask, \ldots, d_m \wedge mask)$
6 $\quad avg \leftarrow \frac{1}{2^n} \sum_{k=1}^{m} 2^{k-1} \cdot \#ON(\hat{d}_k)$
7 $\quad low \leftarrow low + (2^i - 1)^{-1} \cdot avg$
8 $\quad up \leftarrow up + (2^{i-1})^{-1} \cdot avg$
9 **return** (low, up)

The relative width of the interval produced by Algorithm 1 is at most 2, i.e. ${}^{up}/_{low} \leq 2$.

V. Greedy Bucket-based BDD Minimization

BDD minimization, in general and in approximate settings, using genetic algorithms is a common approach, see, e.g., [7, 13, 16]. The optimization problem inherently has a multi-dimensional target space, minimizing both the BDD size and the error introduced through the approximation. While this makes genetic algorithms an appropriate choice, properly designed greedy algorithms can compete as well. By quantizing the space of possible error metric values, a single smallest function approximation can be stored for each possible error metric value tuple. For simplicity, the following description will only apply to exactly one error metric. The method can easily be extended to multiple error metrics.

Algorithm 2 depicts the proposed method. Given a maximum acceptable error metric value m_e, the algorithm splits the range of error values into b buckets of equal size holding one approximated function with an error in that range. Initially, all buckets are initialized with the original function f.

The algorithm works by iteratively applying approximation operations from a set of allowed operations $\mathbb{A}$, e.g. rounding-

down on all levels, to the function f. We support all approximation operations defined in [8].

For each of these operations, the error metric is calculated (line 8) and the matching bucket with index nb for the error rate is determined (line 10). This computation is an implicit check whether the threshold m_e is reached: if nb is larger than the number of allocated buckets, the error is too large.

If the error is in the accepted range and the newly created function is smaller than the one currently in the bucket for this error range (line 11), the function in the bucket is replaced (line 12). Then, the bucket storing the currently used function, cb, is greedily updated to refer to the function with the smallest error metric value (line 13).

This process is repeated until all operations have been applied. Afterwards, if the current bucket cb is larger than the number of available buckets (i.e. the smallest error that can be introduced is larger than the threshold), the algorithm terminates.

The proposed method implicitly applies all ideas presented in Section IV to reduce the run time. We omit the details for didactic reasons.

After the algorithm has finished, buckets still containing the initial function f are discarded. The remaining buckets then form the computed Pareto front.

Algorithm 2: Greedy, Bucket-based BDD Approximation

input : Function f, approximation operations $\mathbb{A}$
input : Error metric em
input : Error threshold m_e, number of buckets b
output: Pareto Front for $\mathrm{size}(\hat{f})$ and $\mathrm{em}(f, \hat{f})$
▷ *Initialize all buckets with original function*
1 $bucket_i \leftarrow f$ for $1 \leq i \leq b$
2 $cb \leftarrow 1$ ▷ *starting bucket*
3 **while** $cb \leq b$ **do** ▷ *Within buckets with error below* m_e
4 $d \leftarrow bucket_{cb}$ ▷ *Smallest function in buckets*
5 $cb \leftarrow cb + 1$
6 **foreach** *approx. operation* $approx \in \mathbb{A}$ **do**
7 $\hat{d} \leftarrow approx(d)$
8 $e \leftarrow \mathrm{em}(f, \hat{d})$
9 $c \leftarrow \mathrm{size}(\hat{d})$
10 $nb \leftarrow \left\lfloor \frac{e \cdot b}{m_e} \right\rfloor + 1$ ▷ *Bucket (i.e. error) for* $\hat{d}$
▷ *Error below threshold and size reduction?*
11 **if** $nb \leq b$ *and* $c < \mathrm{size}(bucket_{nb})$ **then**
12 $bucket_{nb} \leftarrow \hat{d}$
13 $cb \leftarrow min\{cb, nb\}$
14 **return** $(bucket_1, \ldots, bucket_b)$

Example 3: We illustrate the method using the data in Table II with the error rate as the used metric. The upper bound for the error rate is 5%. The range from 0% to 5% error rate is divided into five buckets. Next to the function, the corresponding BDD size is shown.

The first selection step (1 (s)) selects the function stored in bucket 1 (highlighted in bold). To this function, all operations from the set $\mathbb{A}$ are applied (lines 3–13 in Algorithm 2), yielding two new functions $\hat{f}_1$ and $\hat{f}_2$ that are within the error threshold and smaller in size than f (see step 1(a)). They are placed in bucket 1 and 2 respectively.

The decisions in lines 11 and 13 of Algorithm 1 pick bucket 1 for the next step (step 2(s)). After applying all operations from $\mathbb{A}$, the new functions $\hat{f}_3$ and $\hat{f}_4$ are added to the corresponding buckets (step 2(a)). The algorithm then continues with bucket 2 (step 3(s)), even though the function stored in it was not added in the previous step. This process continues until there are no buckets left to pick unapproximated functions from.

In this example, this situation is reached after ten iterations. The Pareto front for the objectives BDD size and error rate are exactly the functions in the buckets in the last line of Table II.

TABLE II: Example run for Algorithm 2.

step	0%–1% $bucket_1$		1%–2% $bucket_2$		2%–3% $bucket_3$		3%–4% $bucket_4$		4%–5% $bucket_5$	
1 (s)	$\mathbf{f}$	280	f	280	f	280	f	280	f	280
1 (a)	$\hat{f}_1$	247	$\hat{f}_2$	256						
2 (s)	$\mathbf{\hat{f}_1}$	247	$\hat{f}_2$	256	f	280	f	280	f	280
2 (a)					$\hat{f}_4$	196			$\hat{f}_3$	150
3 (s)	$\hat{f}_1$	247	$\mathbf{\hat{f}_2}$	256	$\hat{f}_4$	196	f	280	$\hat{f}_3$	150
⋮	⋮		⋮		⋮		⋮		⋮	
10 (a)	$\hat{f}_1$	247	$\hat{f}_{15}$	171	$\hat{f}_{12}$	130	$\hat{f}_{21}$	111	$\hat{f}_{23}$	109

The presented approach can trivially be extended to compute a Pareto front containing multiple error metrics. Instead of a list of buckets, a multi-dimensional grid of buckets can be used. The computation of nb is then performed individually for each error metric.

VI. Experimental Results

We implemented all methods and algorithms presented in this paper in C++ using the CUDD framework [17] for manipulating BDDs. To facilitate reproduction and further works, we publish our framework on github[3]. The experiments were performed on an Intel® Core™ i5-4200U CPU @ 1.60 GHz running Linux (Ubuntu 18.04). Proper measures were taken to avoid contaminating timing measurements with the caches employed by CUDD. To provide accurate millisecond timings, the google benchmark library[4] was used.

As examples for real world functions, the ISCAS'85 dataset [18] and approximate adders from the KIT open-source Approximate Adder Library [19] were used.

A. Average-Case Error Computation

We evaluated the computation of the average-case error when using the new improved ON-set based algorithm. The characteristic function based computation from [8] is used as a baseline. Table III shows the computation time of these two

[3]https://github.com/keszocze/abo
[4]https://github.com/google/benchmark

TABLE III: Comparison of computation time for characteristic function and ON-set based average-case error in milliseconds.

Circuit	Operator	Previous [8]	Proposed	Speedup
c432	$\lceil f \rceil_{x_{35}}$	81	11	7
c499	$\lfloor f \rfloor_{x_{36}}$	4602	43	107
c1908	$\lceil f \rceil_{x_{27}}$	122835	23	5341

algorithms when applied to approximated circuits from the ISCAS'85 dataset. It can be seen that the proposed method is faster than the previous one based on characteristic functions.

While for smaller functions the speed improvement is around a factor of ten, it increases with increasing number of nodes in a function. This result is expected as the proposed average value computation examines every node in the BDD exactly once and thus runs in linear time. In comparison, the characteristic function based implementation may construct an exponential number of new nodes.

B. Approximating the Average-Case Relative Error

To justify approximating the average-case relative error, we compare the computation time and accuracy of the approximate versions with exact computation methods. For computing the average-case relative error exactly, so far only the ADD based computation introduced in [9] is known.

Table IV shows the results of our approximate computation when compared to the reference implementation [9]. Both approaches were run on a selection of approximate adders taken from the open-source KIT library [19]. The table presents the circuit name, the exact error and our approximated error, the relative error, the runtimes and speedup achieved by our approximation method.

The speedup ranges from around five to ten for small 8-bit adders to roughly $23,000$ for 16-bit adders. At the same time, the relative error never exceeds 0.05 which is within an acceptable range. For larger circuits, the relative error cannot be calculated as the reference implementation times out after 60 minutes and, therefore, no values to compare against could be obtained.

C. Greedy Bucket-based BDD Minimization

1) Single-Error-Metric Optimization: To demonstrate the effectiveness of the proposed greedy BDD minimization procedure, we ran it on different functions from the ISCAS'85 dataset. As a baseline comparison, the genetic optimization framework pagmo2 [20] was used, running NSGA-II [21] to jointly minimize BDD size and error metrics. Both algorithms were given the same set of approximation operations to select from and were allowed the same amount of computation time.

Table V shows the results of running the NSGA-II-based algorithm and our proposed approach. As the benchmarks aren't arithmetic functions, the error rate was chosen as the error metric and the maximum accepted value for the error rate was set to 5%. A total of 15 buckets was used. We report the smallest function found below the threshold of 5% error rate. As the set of possible approximation operations, all operations introduced in this paper were used. They were applied to every variable independently for every sub-function f_i.

It can be seen that the proposed method performs well, with respect to runtime as well as to the objectives error rate and BDD size. Except for the c3540 function, the proposed method runs faster than the NSGA-II approach. The speedup is within the range of 1 to 10 with comparable error rates. As can be seen, the BDD sizes produced by the proposed method never exceed the NSGA-II results while improvements up to using only 70% of the NSGA-II size can be reported.

2) Multi-Error-Metric Optimization: To further demonstrate the applicability of the proposed BDD minimization framework, we present results for optimizing the size as well as the two error metrics error rate and average-case error at once in Table VI. In this experiment, we aim to compare ourselves against the adders from [19]. For each error metric, the number of buckets is 10 and the threshold for the error rate is 5% in all experiments. The threshold on the average-case error is set to $7, 35$, and 9000 for $8, 16$, and 32 bit, respectively. In comparison to the NSGA-II-based approach, the proposed method sill runs ≈ 5.5 times faster on average. The resulting functions are comparable to the results from [19]; for 8-bit adders, in fact, identical functions were found by our approach.

VII. Conclusion

In this paper we presented multiple means for the acceleration of error metric computations as well as a greedy, bucket-based BDD minimization approach exploiting these accelerations. The experiments clearly show that considerable speedup were achieved while, at the same time, Pareto front for multiple error metrics close to results from state-of-the-art approximate logic synthesis approaches.

To allow for further research by interested scientists, we publish our framework as open-source on github.

Acknowledgment

We would like to thank the authors of [9] for providing us with a binary of their tool.

References

[1] Ning Zhu et al. "Design of Low-Power High-Speed Truncation-Error-Tolerant Adder and Its Application in Digital Signal Processing". In: *Transactions on Very Large Scale Integration Systems* 18.8 (Aug. 2010).

[2] Rangharajan Venkatesan et al. "MACACO: Modeling and Analysis of Circuits for Approximate Computing". In: *International Conference On Computer Aided Design*. 2011.

[3] Jie Han and Michael Orshansky. "Approximate Computing: An Emerging Paradigm for Energy-Efficient Design". In: *European Test Symposium*. May 2013.

[4] Sparsh Mittal. "A Survey of Techniques for Approximate Computing". In: *ACM Comput. Surv.* 48.4 (Mar. 2016).

[5] Swagath Venkataramani et al. "Approximate Computing and the Quest for Computing Efficiency". In: *Design Automation Conference*. 2015.

[6] Vojtech Mrazek et al. "EvoApprox8B: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods". In: *Design, Automation and Test in Europe*. 2017.

[7] Zdenek Vasicek and Lukas Sekanina. "Evolutionary Design of Complex Approximate Combinational Circuits". In: *Genetic Programming and Evolvable Machines* 17.2 (June 1, 2016).

[8] Mathias Soeken et al. "BDD Minimization for Approximate Computing". In: *Asia and South Pacific Design Automation Conference*. Jan. 2016.

TABLE IV: Comparing the values for the average-case relative error and the runtime of the approach from [9] to our proposed average-case relative error approximation. The values for acre, $\hat{\text{acre}}$, and the runtimes are rounded for readability.

Circuit	acre [9]	$\hat{\text{acre}}$	$\lvert\text{acre} - \hat{\text{acre}}\rvert / \text{acre}$	Time [9]$[ms]$	Time ours$[ms]$	Speedup
ACAI_N8_Q5	1.1	1.2	0.008	2.5	0.47	5.3
ACAII_N8_Q4	0.16	0.15	0.02	1.6	0.59	2.7
GeAr_N8_R1_P5	0.26	0.25	0.026	2.6	0.44	5.9
GeAr_N8_R2_P2	0.16	0.15	0.02	4.3	0.52	8.3
ACAI_N16_Q4	3	3.1	0.05	54772	5.1	10740
ACAII_N16_Q4	1.8	1.9	0.042	53589	2.2	24359
ACAII_N16_Q8	1.5	1.6	0.047	46448	2.2	21113
ACAI_N32_Q8	n/a	3.1	n/a	$timeout$	32	n/a
ACAII_N32_Q16	n/a	0.91	n/a	$timeout$	5.9	n/a

TABLE V: Comparing the proposed greedy, bucket-based BDD minimization method to a default implementation of an evolutionary algorithm using NSGA-II optimizing for error rate and BDD size.

		NSGA-II			Proposed				
Circuit	$size(f)$	$size(\hat{f})$	er(%)	Time	$size(\hat{f})$	er(%)	Time	Speedup	$\frac{size(proposed)}{size(NSGA-II)}$
c432	1223	1115	3.9	$2s$	1063	4.8	$2s$	1	0.95
c499	26808	17203	4.7	$208m$	12691	4.9	$2m$	104	0.74
c880	8489	2278	5	$11m$	1630	5.0	$5m$	2.2	0.72
c1355	40202	34482	4.9	$40m$	23115	4.5	$4m$	10	0.67
c1908	12448	225	1.2	$415s$	225	1.2	$39s$	4.7	1.00
c2670	5625	3874	4.9	$151s$	3865	4.8	$83s$	1.6	> 0.99
c3540	122664	92600	4.9	$25m$	82620	4.9	$37m$	0.7	0.89

TABLE VI: Comparing the proposed method to an evolutionary algorithm using NSGA-II optimizing for BDD size and multiple error metrics (error rate and average-case error) for adders of different bit-width.

			NSGA-II				Proposed				
Bits	m_{ace}	$size(f)$	$size(\hat{f})$	$ER(\%)$	$ace(f,\hat{f})$	Time	$size(\hat{f})$	$ER(\%)$	$ace(f,\hat{f})$	Time	Speedup
8	7	116	89	4.7	3.5	$7s$	89	4.7	3.5	$1s$	7
16	35	424	288	4.3	34.8	$3m$	286	4.7	15.5	$35s$	6
32	9000	1616	992	4.3	8702	$183m$	965	3.0	1610	$47m$	3.9

[9] Saman Froehlich, Daniel Große, and Rolf Drechsler. "One Method - All Error-Metrics: A Three-Stage Approach for Error-Metric Evaluation in Approximate Computing". In: *Design, Automation and Test in Europe*. Mar. 2019.

[10] Oliver Keszocze, Mathias Soeken, and Rolf Drechsler. "The Complexity of Error Metrics". In: *Information Processing Letters* 139 (Nov. 2018).

[11] Honglan Jiang, Jie Han, and Fabrizio Lombardi. "A Comparative Review and Evaluation of Approximate Adders". In: *Great Lakes Symposium on VLSI*. 2015.

[12] Milan Češka et al. "Approximating Complex Arithmetic Circuits with Formal Error Guarantees: 32-Bit Multipliers Accomplished". In: *International Conference On Computer Aided Design*. 2017.

[13] Saeideh Shirinzadeh et al. "An Adaptive Prioritized E-Preferred Evolutionary Algorithm for Approximate BDD Optimization". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. 2017.

[14] Jorge Echavarria, Stefan Wildermann, and Jürgen Teich. "Design Space Exploration of Multi-Output Logic Function Approximations". In: *International Conference On Computer Aided Design*. 2018.

[15] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Transactions on Computers* C-35.8 (Aug. 1986).

[16] Shan-Tai Chen et al. "Towards the Exact Minimization of BDDs—An Elitism-Based Distributed Evolutionary Algorithm". In: *Journal of Heuristics* 10.3 (May 1, 2004).

[17] Fabio Somenzi. *CUDD: CU Decision Diagram Package*.

[18] Mark C. Hansen, Hakan Yalcin, and John P. Hayes. "Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering". In: *IEEE Design & Test of Computers* 16.3 (1999).

[19] Muhammad Shafique et al. "A Low Latency Generic Accuracy Configurable Adder". In: *Design Automation Conference*. https://sourceforge.net/projects/approxadderlib/. 2015.

[20] Francesco Biscani et al. *Pagmo*. Version 2.7. https://doi.org/10.5281/zenodo.1217831.

[21] Kalyanmoy Deb et al. "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II". In: *IEEE Transactions on Evolutionary Computation* 6.2 (Apr. 2002).