BDD Minimization for Approximate Computing

Mathias Soeken[1,2] Daniel Große[2] Arun Chandrasekharan[2] Rolf Drechsler[2,3]
[1]Integrated Systems Laboratory (LSI), EPFL, Switzerland
[2]Faculty of Mathematics and Computer Science, University of Bremen, Germany
[3]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
{msoeken,grosse,arun,drechsle}@cs.uni-bremen.de

Abstract—We present *Approximate BDD Minimization* (ABM) as a problem that has application in approximate computing. Given a BDD representation of a multi-output Boolean function, ABM asks whether there exists another function that has a smaller BDD representation but meets a threshold w.r.t. an error metric. We present operators to derive approximated functions and present algorithms to exactly compute the error metrics directly on the BDD representation. An experimental evaluation demonstrates the applicability of the proposed approaches.

I. Introduction

Approximate computing refers to techniques that relax the requirement of exact equivalence between the specification and the implementation [19]. There exist several applications, such as media processing (audio, video, graphics, and image), recognition, and data mining, that tolerate acceptable though not always correct results [9]. Due to inherent error resilience a precise functional behavior is not required. Several factors such as the limited perceptual capability of humans allow imprecision in the numerical exactness of the computation in these applications. This freedom can be exploited in the implementation of the applications by achieving significant improvements in terms of performance and energy efficiency in comparison to the exact implementation [2, 4, 15]. Several approaches for mathematical analyses related to imperfect computation have been proposed which are heavily used in the implementation of approximate computing [2].

Research in approximate computing spans the whole range of research activities ranging from programming languages [6] to transistors [8]. There exist two main strategies to introduce imperfection to the circuit with the aim to improve its performance [19]: (i) timing-induced errors, e.g., by voltage over-scaling or over-clocking, and (ii) functional approximation, e.g., by implementing a slightly different function. Our research targets the latter strategy. Given a specification $f : \mathbb{B}^n \to \mathbb{B}^m$ that describes the correct functionality, an approximated function $\hat{f} : \mathbb{B}^n \to \mathbb{B}^m$ is sought that minimizes a given circuit cost metric but respects a quality threshold.

In this paper, we introduce the *Approximate BDD Minimization* (ABM) problem. It asks whether for a given multi-output function represented as a *Binary Decision Diagram* (BDD), a given threshold based on some error metric, and a size bound, a new function can be obtained that (i) is an approximation to the original function with respect to the threshold, and (ii) has a BDD representation whose size does not exceed the bound. For function representation and manipulation BDDs have become state-of-the-art [3]. They have been studied in logic synthesis, since they allow to combine aspects of circuit synthesis and technology mapping. There has been a renewed interest in multiplexor-based design styles (e.g., [20, 21]), since multiplexor nodes can often be realized at very low cost (e.g., pass transistor logic (PTL)). In addition, these techniques allow us to consider layout aspects during the synthesis step and therefore guarantee high design quality (see, e.g., [11, 12]). In this context, circuits derived from BDDs often result in smaller netlists.

For solving the ABM problem this paper makes the following three key contributions:

1. presentation of a generic algorithm for ABM,
2. presentation of five symbolic BDD approximation operators, and
3. development of symbolic algorithms for the exact computation of frequently used error metrics.

In the experiments we evaluate the effectivity of the approximation operators and the performance of the symbolic algorithms for the exact error metric computation.

II. Related Work

A. Approximate Computing

In [18] a systematic logic synthesis methodology has been proposed that maps the problem of approximate synthesis into a classical logic synthesis problem. Consequently, the capabilities of existing synthesis tools can be fully utilized for approximate logic synthesis. A simple two-level synthesis approach is described in [16] that aims at minimizing circuit area for a given error rate threshold. The approach is rather simple and is not optimized for multiple-output functions. It has been extended for multi-level synthesis in [17]. Again, the approach aims at minimizing circuit area by respecting a given *rate significance* threshold, which is a composite metric based on worst-case error and error rate. The algorithm is based on automatic test pattern generation methods and relaxes the definition of a redundant fault to minimize circuit area.

B. BDD Minimization

Most related to the proposed research is the minimization of incompletely specified functions, i.e., for some values x we *don't care* whether $f(x) = 1$ or $f(x) = 0$. Incompletely specified functions can be represented by a Boolean function f and a Boolean function g, called *don't care set*, such that $f(x)$ is a don't care value if $g(x) = 0$. The minimization problem is to find a function $\hat{f}$, called *cover*, whose BDD representation has a small number of nodes, referred to as $B(\hat{f})$, such that

$$f(x) \wedge g(x) \leq \hat{f}(x) \leq f(x) \vee \bar{g}(x) \qquad \text{for all } x. \tag{1}$$

In other words, $\hat{f}(x)$ must agree with $f(x)$ whenever x satisfies $g(x) = 1$, but we don't care what value $\hat{f}(x)$ assumes when $g(x) = 0$.

The associated decision problem is called *Exact BDD Minimization* (EBM) and asks for a given function f, a don't care

set g, and a size bound b, whether there exists a function $\hat{f}$ as in Eq. (1) such that $B(\hat{f}) \leq b$. The authors in [14] have proven that deciding EBM is NP-complete. Further, they have also shown that efficient approximation algorithms for EBM exist only if NP = P.

III. Preliminaries

A. Error Metrics

The quality of an approximated circuit is evaluated using multiple error metrics. The *worst-case error*

$$\mathrm{wc}(f, \hat{f}) = \max\{|\operatorname{int}(f(x)) - \operatorname{int}(\hat{f}(x))| \; \forall x \in \mathbb{B}^n\}, \quad (2)$$

where 'int' returns the integer representation of a bit vector, is the maximum difference between the approximate output value and a correct version for all possible inputs. This metric is sometimes also referred to as *error significance* in the literature. The *average-case error*

$$\mathrm{ac}(f, \hat{f}) = \frac{\sum_{x \in \mathbb{B}^n} |\operatorname{int}(f(x)) - \operatorname{int}(\hat{f}(x))|}{2^n} \quad (3)$$

is the average difference and the *error rate*

$$\mathrm{er}(f, \hat{f}) = \frac{\sum_{x \in \mathbb{B}^n} [f(x) \neq \hat{f}(x)]}{2^n} \quad (4)$$

is the ratio of the errors observed in the output value as a result of approximation to the total number of input combinations. The product of error rate and the total number of input combinations corresponds to the Hamming distance when applied to single-output functions. The first two metrics consider the integer representation of the function values instead of the binary representation. This is useful since in most applications a change in more significant bits has a much higher effect than changes in less significant bits.

Throughout the paper, the multiple-output functions f and $\hat{f}$ are represented as m-tuples $(f_{m-1}, \ldots, f_0)$ and $(\hat{f}_{m-1}, \ldots, \hat{f}_0)$.

B. Binary Decision Diagrams

A BDD (e.g., [5]) is a graph-based representation of a function that is based on the Shannon decomposition $f = x_i f_{x_i} \oplus \bar{x}_i f_{\bar{x}_i}$. Applying this decomposition recursively allows dividing the function into many smaller sub-functions. Solid and dashed lines refer to high and low successors, respectively. BDDs make use of the fact that for many functions of practical interest, smaller sub-functions occur repeatedly and need to be represented only once. Combined with an efficient recursive algorithm that makes use of caching techniques and hash tables to implement elementary operations, BDDs are a powerful data structure for many practical applications. BDDs are ordered in the sense that the Shannon decomposition is applied with respect to some given variable ordering which also has an effect on the BDD's number of nodes. Improving the variable ordering for BDDs is NP-complete [1] and many heuristics have been presented that aim at finding a good ordering. Throughout this paper, we only consider BDDs with a fixed variable ordering and we assume that this is the natural one $x_1 < x_2 < \cdots < x_n$.

Given a Boolean function $f(x) = f(x_1, \ldots, x_n)$, $|f|$ refers to the number of binary vectors $x = x_1 \ldots x_n$ such that $f(x) = 1$ (ON-set). We define $B(f)$ to be the size of the BDD representation for f and a given variable ordering, which is the total number of nodes, including the sinks.

C. Characteristic Functions

Let $f : \mathbb{B}^n \to \mathbb{B}^m$ be a multi-output Boolean function with $f(x_1, \ldots, x_n) = (f_{m-1}, \ldots, f_0)$. Its characteristic function $\chi_f : \mathbb{B}^{m+n} \to \mathbb{B}$ is a single-output Boolean function and is defined as follows:

$$\chi_f(x_1, \ldots, x_n, y_{m-1}, \ldots, y_0) = \bigwedge_{0 \leq j < m} (f_j(x_1, \ldots, x_n) \oplus \bar{y}_j), \quad (5)$$

by forcing y_j to equal the evaluation of $f_j(x_1, \ldots, x_n)$. The characteristic function evaluates to true, if and only if $x_1, \ldots, x_n$ and $y_{m-1}, \ldots, y_0$ are a valid input/output pattern for f.

IV. Approximate BDD Minimization

Central to our research is the *Approximate BDD Minimization* (ABM) problem which is presented for the first time in this paper. ABM aims at minimizing a BDD representing a given Boolean function f by approximating it with a Boolean function $\hat{f}$ and respecting a given threshold based on an error metric. The associated decision problem asks for a given function f, an error metric e, a threshold t, and a size bound b, whether there exists a function $\hat{f}$ such that $e(f, \hat{f}) \leq t$ and $B(\hat{f}) \leq b$.

In this paper, we consider the frequently used worst-case error, average-case error, and error rate as error metrics e. In the following we give a *non-deterministic generic algorithm* to solve ABM.

Algorithm A (*Approximate BDD Minimization*). The algorithm gets as input a Boolean function f, an error metric e, a threshold t, and a size bound b.

A1. [Initialize.] Set $\hat{f} \leftarrow f$.

A2. [Is $\hat{f}$ small enough?] If $B(\hat{f}) \leq b$, return $\hat{f}$ and terminate.

A3. [Approximate $\hat{f}$.] Set $h \leftarrow \texttt{APPROX}(\hat{f})$.

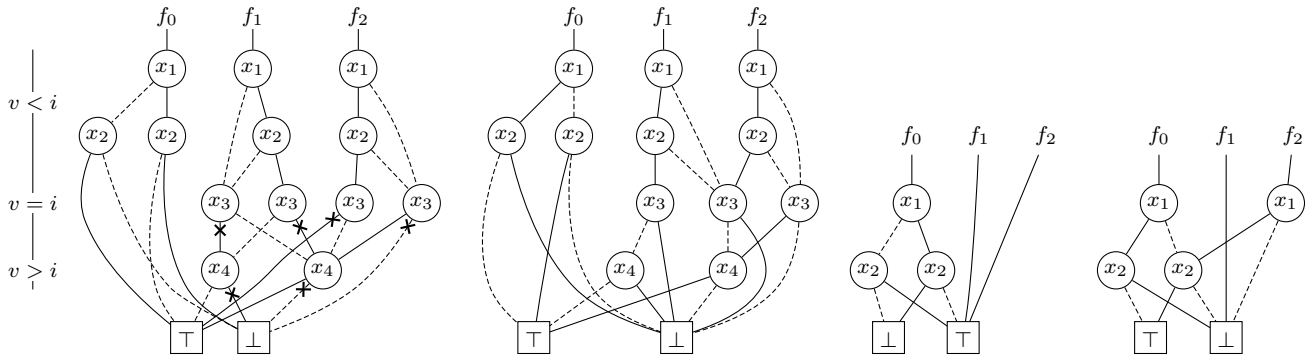
A4. [Evaluate error metric.] If $e(f, h) \leq t$ and $B(h) < B(\hat{f})$, set $\hat{f} \leftarrow h$, and return to step 2; otherwise return to step 3.

The function `APPROX` in step 3 of Algorithm A refers to some BDD operator that approximates $\hat{f}$ further (potentially controlled by user preferences). This result is stored in the temporary variable h. Before $\hat{f}$ can be replaced by h, it needs to be checked whether (i) h has a smaller BDD representation than $\hat{f}$ and (ii) the error metric respects the given threshold. The algorithm can be made deterministic by (i) providing a strategy that selects approximation operators in step 3 and (ii) by relaxing the condition in step 2 to reach the size bound b but terminate already beforehand. The latter condition guarantees that the algorithm completes, however, a heuristic solution may be returned.

Although Algorithm A is simple and generic, its non–trivial parts are the approximation operator `APPROX` and the computation of the error metric e directly on the BDD representation. Solutions to both of these are presented in the following two sections. Section V presents five operators that can be used to approximate a function in its BDD representation, and Section VI shows algorithms to compute each of the three error metrics symbolically on the BDD representation.

V. Approximation Operators

This section describes five approximation operators which are summarized in Table I. Applying the operator to a multi-output function denotes applying it to each sub-function.



(a) Rounding example f (b) Rounding down $\lfloor f \rfloor_{x_3}$ (c) Rounding up $\lceil f \rceil_{x_3}$ (d) Rounding $[f]_{x_3}$

Fig. 1. Example for approximation operators

TABLE I
Approximation operators

Op.	Description	Op.	Description
f_{x_i}	Positive co-factor	$\lfloor f \rfloor_{x_i}$	Rounding down
$f_{\bar{x}_i}$	Negative co-factor	$\lceil f \rceil_{x_i}$	Rounding up
		$[f]_{x_i}$	Rounding

Co-factor approximation. One of the simplest approximation operators is taking the co-factor with respect to some variable x_i, i.e.,

$$\hat{f} \leftarrow f_{x_i} \quad \text{or} \quad \hat{f} \leftarrow f_{\bar{x}_i}. \tag{6}$$

Approximation by rounding. We define two operators $\lfloor f \rfloor_{x_i}$ and $\lceil f \rceil_{x_i}$ for *rounding up* and *rounding down* a function based on the BDD. The idea is inspired by [13]: for each node that appears at level x_i or lower (in other words for each node labeled x_j with $j \geq i$), the lighter child, i.e., the child with the smaller ON-set, is replaced by a terminal node. The terminal node is $\bot$ when rounding down, and $\top$ when rounding up. The technique is called *heavy branch subsetting* in [13].

Example 1 *Fig. 1(a) shows a BDD for a function with four inputs and three outputs which serves as a running example throughout this section. Each example applies rounding at level* 3 *and for rounding up and rounding down, crosses emphasize lighter children. Figs. 1(b) and (c) show the resulting BDDs after applying rounding down and rounding up, respectively.*

The algorithms for rounding down and rounding up do not necessarily reduce the number of variables since only one child is replaced by a terminal node. The last approximation operator *rounding* does guarantee a reduction of the number of variables since it replaces all nodes of a given level by a terminal node. Which terminal node is chosen depends on the size of the ON-set of the function represented by that node. If the size of the ON-set ($|f|$) exceeds the size of the OFF-set ($|\bar{f}|$), the node is replaced by $\top$, otherwise by $\bot$.

Example 2 *Figs. 1(d) shows the effect of rounding at level* 3.

VI. Computing Error Metrics with BDDs

This section shows how to compute the error metric symbolically on the BDD represenation of f and $\hat{f}$.

Error rate counts for the percentage for how many input assignments the function value differs. Since a function value differs if at least one sub-function is different, error rate can be computed using the formula

$$\mathrm{er}(f, \hat{f}) = \frac{1}{2^n} \left| \bigvee_{0 \leq i < m} f_i(x) \oplus \hat{f}_i(x) \right|. \tag{7}$$

For the other two error metrics the outputs of a multi-output Boolean function are assumed to represent a natural number or integer $k = f(x)$. In case of a natural number, function f_i represents the i^{th} bit in the binary expansion of k, i.e.,

$$f(x) = \sum_{0 \leq i < m} f_i(x) \cdot 2^i. \tag{8}$$

For integers, we use two's complement

$$f(x) = -f_{m-1}(x) \cdot 2^{m-1} + \sum_{0 \leq i < m-1} f_i(x) \cdot 2^i. \tag{9}$$

Worst-case error and average-case error have the expression $d(x) = |\operatorname{int}(f(x)) - \operatorname{int}(\hat{f}(x))|$ in common. It returns the absolute value difference at input x. Subtraction is calculated using m full-adders with 1 as first carry-in and inverting each bit of the subtrahend. The bit fiddling trick from [7] is used to compute the absolute value.

For the *worst-case error* the maximum value must be computed, for which we present two algorithms.

Algorithm M (*Maximum value*). Given m BDDs $(d_{m-1}, \ldots, d_0)$ that represent a mapping $d(x_1, \ldots, x_n)$ as in (8), this algorithm computes $v = \max d(x_1, \ldots, x_n)$. It also computes a Boolean function μ with $\mu(x_1, \ldots, x_n) = 1$ if and only if $d(x_1, \ldots, x_n) = v$.

M1. [Initialize.] Set $v \leftarrow 0$, $i \leftarrow m$, and $\mu \leftarrow \top$.

M2. [Terminate?] If $i = 0$, terminate, otherwise set $i \leftarrow i - 1$.

M3. [Compute mask.] Set $\mu' \leftarrow \mu \wedge f_i$.

M4. [Update v and μ?] If $\mu' \neq \bot$, set $v \leftarrow v + 2^i$ and $\mu = \mu'$. Return to step 2.

Example 3 *We illustrate Algorithm M using the following example:*

x_2	x_1	d_4	d_3	d_2	d_1	d_0	k
0	0	0	1	0	1	0	10
0	1	0	0	1	1	0	6
1	0	0	1	1	0	1	13
1	1	0	1	1	0	0	12

(10)

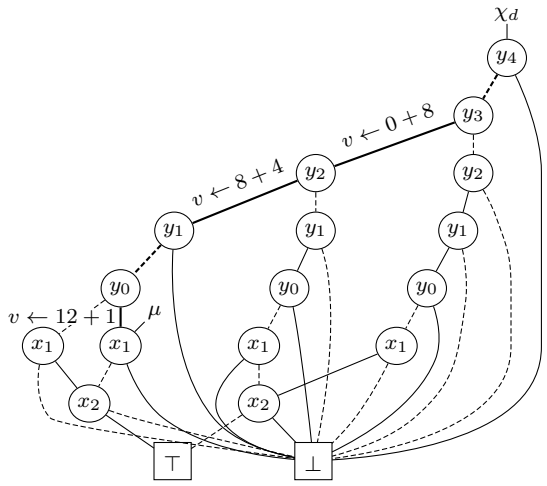


Fig. 2. Characteristic function of (10) to compute the maximum value

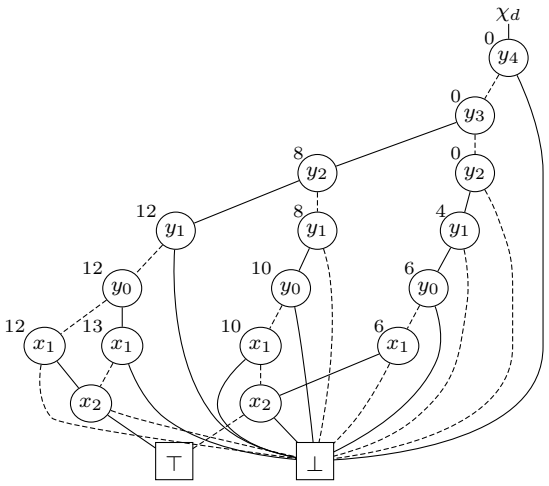


Fig. 3. Characteristic function of (10) to compute the weighted sum

The values in the algorithm then change as follows (for brevity only the steps are listed in which at least one of the values for i, v, or μ changes):

Step	i	v	μ
M1	5	0	1111
M2	4	0	1111
M2	3	0	1111
M4	3	8	1011
M2	2	0	1011

Step	i	v	μ
M4	2	12	0011
M2	1	12	0011
M2	0	12	0011
M4	0	13	0010

The maximum value is $v = 13$ and we have $\mu = \bar{x}_1 x_2$. Note that μ is given as a bit-string that represents its truth table with the leading bit referring to the most-significant bit.

Algorithm M iterates over the columns of the truth table; it needs to perform m BDD operations. Input assignments that do not evaluate to the maximum value are ruled out early using the mask μ. The algorithm cannot be modified to compute the average-case error since all function values need to be considered and not only the maximum one. That would require to traverse all 2^n rows of the truth table instead. To overcome this limitation, we are following an alternative idea that uses the characteristic function of $d(x)$. But before we describe the algorithm, we first give an alternative algorithm to find the maximum value based on the same idea we use for the approximate-case error. This facilitates the understanding.

Algorithm N (*Maximum value*). Given m BDDs $(d_{m-1}, \ldots, d_0)$ that represent a mapping $d(x_1, \ldots, x_n)$ as in (8), this algorithm computes $v = \max d(x_1, \ldots, x_n)$. It also computes a Boolean function μ with $\mu(x_1, \ldots, x_n) = 1$ if and only if $d(x_1, \ldots, x_n) = v$.

N1. [Initialize.] Set $v \leftarrow 0$ and compute the BDD for χ_d with order

$$y_{m-1} < y_{m-2} < \cdots < y_0 < x_1 < \cdots < x_n$$

and let μ be the root node of that BDD.

N2. [Terminate?] If $\mu_v \geq m + 1$, i.e., the variable is labeled $x_{\mu_v - m}$, terminate.

N3. [Next child.] If $\mu_h \neq \bot$, set $\mu \leftarrow \mu_h$ and $v \leftarrow v + 2^{m-\mu_v}$; otherwise, set $\mu \leftarrow \mu_l$. Return to step 2.

Example 4 *Fig. 2 shows the characteristic function of* (10) *(see Example 3.) Starting from the root node, the algorithm follows high edges as long as they do not lead to the zero terminal. Whenever a high edge can be followed the current value of v is incremented by a power of 2 that corresponds to the current level; starting from 2^{m-1} and then decreasing by 1 with every level. Fig. 2 emphasizes this path with thicker lines. The algorithm stops when the first node is encountered that is labeled by an input variable. This node represents μ, i.e., the function of all input assignments that evaluate to the maximum value v.*

Algorithm N can easily be modified to an algorithm that is key to compute the average-case error. For this purpose, we first note that numerator can be written $\sum_{x \in \mathbb{B}^n} d(x)$ and then rewrite it to

$$\sum_{x \in \mathbb{B}^n} d(x) = \sum_{0 \leq v < 2^m} v \cdot |\{x \in \mathbb{B}^n \mid d(x) = v\}|, \qquad (11)$$

i.e., instead of summing up all function values by iterating over all assignments, we iterate over all function values and multiply them with the occurrence of assignments that evaluate to them. We call (11) a *weighted sum* and give an algorithm to compute it using the BDD of χ_d.

Algorithm W (*Weighted sum*). Given m BDDs $(d_{m-1}, \ldots, d_0)$ that represent a mapping $d(x_1, \ldots, x_n)$ as in (8), this algorithm computes $w = \sum_{x \in \mathbb{B}^n} \mathrm{nat}(d(x))$. The auxiliary variables $\texttt{LINK}(g)$ are used to represent a stack of nodes g and a pointer s that points to the top of the stack. Also, the variable $\texttt{VAL}(g)$ represents an integer value whose binary expansion corresponds to the path from the root node of χ_d to node g.

W1. [Initialize.] Set $w \leftarrow 0$ and compute the BDD for χ_d with order

$$y_{m-1} < y_{m-2} < \cdots < y_0 < x_1 < \cdots < x_n$$

and let g be the root node of that BDD. Set $\texttt{LINK}(g) \leftarrow \Lambda$ and $s \leftarrow g$.

W2. [Terminate or pop s?] If S is empty, terminate. Otherwise set $g \leftarrow s$, $s \leftarrow \texttt{LINK}(s)$, and $v \leftarrow \texttt{VAL}(g)$.

W3. [Is g an input node?] If $g_v \geq m + 1$ (i.e., g corresponds to a variable labeled $x_{g_v - m}$), set $w \leftarrow w + |g| \cdot \texttt{VAL}(g)$ and return to step 2.

W4. [g is an output node.] Set $\texttt{LINK}(g_h) \leftarrow s$, $\texttt{LINK}(g_l) \leftarrow g_h$, and $s \leftarrow g_l$. Also, set $\texttt{VAL}(g_l) \leftarrow v$ and $\texttt{VAL}(g_h) \leftarrow v + 2^{m-g_v}$.

TABLE II
SUMMARY OF THE ISCAS-85 BENCHMARK CIRCUITS (FROM [10])

Circuit	Function	#PI	#PO	#Gates	#Blocks
c17	Example circuit	5	2	6	1
c432	27-channel interrupt controller	36	7	160	5
c499	32-bit SEC circuit	41	32	202	2
c1355	32-bit SEC circuit	41	32	546	2
c1908	16-bit SEC/DED circuit	33	25	880	6
c3540	8-bit ALU	50	22	1,669	11

Example 5 *Fig. 3 illustrates Algorithm W applied to the function in* (10)*. Each node is annotated with* $\texttt{VAL}(g)$ *starting from the root note down to the first level of nodes that are labeled by an input variable. Since the function in* (10) *is injective, the weight is always 1, and hence the result is the sum of all computed values* $12+13+10+6 = 41$ *leading to an average-case error of* 10.25.

VII. Experimental Evaluation

We implemented all described approaches in C++ as a command called '`comb_approx`' in the CirKit framework.[1] The experiments were carried out on an Octa-Core Intel Xeon CPU with 3.40 GHz and 32 GB memory running Linux 3.14. The evaluation uses the ISCAS-85 benchmark set. The first experiment evaluates the qualitative effect of the approximation operators and the second experiment evaluates the performance of computing the error metrics.

A. Evaluating approximation operators

We took the combinational benchmark circuits from ISCAS which are listed in Table II. We compared how the relation of the error metric compared to the BDD size evolves when increasing the number of levels in the rounding down, rounding up, and rounding operator (see Table I and Section V). Since the co-factor operators consider one level and do not directly effect the successive ones, they are not part of the evaluation.

The plots in Fig. 4 show the results of this evaluation. The x-axis marks the error rate and the y-axis marks the size improvement of the BDD representation for a particular configuration. The color refers to the approximation operator and a small number above the mark reveals the value for i, i.e., the level at which the operator was applied.

A steep curve means that a high size improvement is obtained by only a small increase in error rate. A flat curve means the opposite: the error rate increases significantly by reducing the BDD only slightly. The circuits 'c17', 'c432', and 'c3540' show neither a steep nor a flat curve. In other words, by rounding more parts of the BDD the size can be reduced by accepting a reasonable increase in the error rate. In 'c1908' the curve is first very steep and then becomes flat, at least for rounding up and rounding. A good trade-off is obtained at an error rate of about 28% and a size improvement of about 92%. The benchmarks 'c499' and 'c1355' show similar (but not as strong) effects. Also it can be noticed that the effects are not as high for rounding down, which gives a more fine grained control over the approximation.

B. Evaluating the computation of error metrics

We evaluated the algorithms to compute the error metric by tracking their run-time. Table III lists the benchmark (column Circuit), the applied approximation operator (column Operator; cf. Table I), and run-times to apply the operator (column Apply) as well as to compute the three error metrics (columns Error Rate, Worst-Case (both algorithms) and Average-Case). For worst-case it is differentiated between the two algorithms to compute the maximum value. Except in two cases the run-times to compute error rate and worst-case using Algorithm M are comparable to the run-times that are required in order to apply the approximation operator. The two exceptions in case of error rate are significant. When using Algorithm N to compute the worst-case, the run-time increases, in some cases by a factor of 100. The run-times to compute the average-case are (often significantly) larger. However, note that the computation exactly determines the average value over, e.g., 2^{41} values in case of 'c499' and 'c1355'.

TABLE III
EVALUATING THE COMPUTATION TIMES OF ERROR METRICS (IN SECONDS)

Circuit	Operator	Apply	ER	WC Alg. M	WC Alg. N	AC
c432	$\lfloor f \rfloor_{x_{35}}$	0.09	0.00	0.01	2.58	2.58
c432	$\lceil f \rceil_{x_{35}}$	0.11	0.02	0.05	2.65	2.70
c432	$[f]_{x_{35}}$	0.01	0.03	0.03	2.66	2.69
c499	$\lfloor f \rfloor_{x_{40}}$	0.45	0.00	0.02	2.58	2.58
c499	$\lfloor f \rfloor_{x_{36}}$	0.45	1.40	0.04	4.00	7.38
c499	$\lceil f \rceil_{x_{40}}$	0.45	248.74	9.56	3.51	252.00
c499	$\lceil f \rceil_{x_{36}}$	0.45	1.20	1.50	4.06	6.78
c499	$[f]_{x_{40}}$	0.00	248.75	0.05	3.51	252.01
c499	$[f]_{x_{36}}$	0.00	0.03	8.77	2.69	2.78
c1355	$\lfloor f \rfloor_{x_{40}}$	0.46	0.00	0.02	2.59	2.58
c1355	$\lceil f \rceil_{x_{40}}$	0.44	248.71	12.32	3.49	251.99
c1908	$\lfloor f \rfloor_{x_{32}}$	0.10	0.00	0.01	2.58	2.59
c1908	$\lfloor f \rfloor_{x_{27}}$	0.11	0.33	0.04	24.36	67.32
c1908	$\lfloor f \rfloor_{x_{23}}$	0.09	0.46	0.06	47.98	137.19
c1908	$\lceil f \rceil_{x_{32}}$	0.10	0.09	0.23	3.14	5.57
c1908	$\lceil f \rceil_{x_{27}}$	0.09	0.17	0.59	31.70	100.39
c1908	$\lceil f \rceil_{x_{23}}$	0.10	0.16	0.57	59.79	203.85
c1908	$[f]_{x_{32}}$	0.01	0.08	0.02	3.14	5.58
c1908	$[f]_{x_{27}}$	0.00	0.32	0.41	24.03	50.14
c1908	$[f]_{x_{23}}$	0.00	0.16	0.74	48.25	109.71

Using the characteristic function to compute the worst-case error is not a good idea, and Algorithm M is a better choice. However, in order to compute the average-case, so far no better algorithm is known and it remains an open question, whether a technique that does not depend on the characteristic function representation exists, which can symbolically (and exact) compute this error metric.

Since the presented approaches depend on BDDs, they are not applicable to much larger circuits in their current implementation. However, the following modifications are possible in order to improve the scalability:

1. Partition the circuit into subcircuits and perform approximation on the individual subcircuits: Additional algorithms and strategies are then required to estimate and control the overall approximation after merging the approximated subcircuits.

2. Change the underlying data structure, e.g., to AND-inverter graphs (AIGs): All algorithms cannot be transferred to AIGs in straight-forward manner, and it is not clear whether exact error metrics can be computed efficiently.

[1] The code can be downloaded from github.org/msoeken/cirkit, see also www.informatik.uni-bremen.de/agra/eng/maniac.php

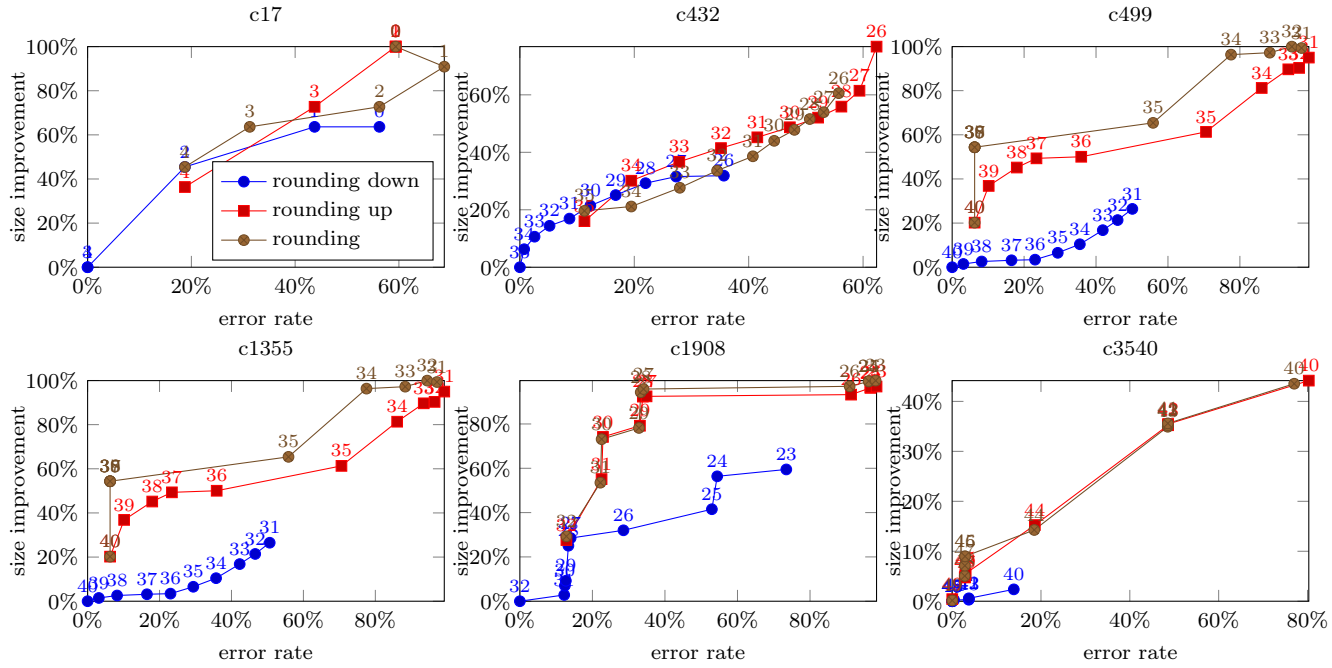


Fig. 4. Evaluating approximation operators

VIII. Conclusions

We introduced the *Approximate BDD Minimization* (ABM) problem in this paper and provided a generic algorithm in order to solve it. The two key parts in this algorithm are the approximation of the function using BDD operations and the computation of the error metric. We proposed five BDD operations and several algorithms in order to compute the error metric directly on the BDD representation. Experiments evaluate both, the effect of the approximation operators and the performance of computing the error metrics.

The paper suggests the applicability of BDDs as a data structure for algorithms in approximate computing. It is shown that important tasks such as *exactly* computing the error metric can be performed symbolically without the need of enumerating all function values or approximating them. This computation can be used independently of Algorithm A and the proposed approximation operators in algorithms for approximate computing that use alternative techniques to derive $\hat{f}$. In future work we plan to consider the combination of the proposed approximation operators. In addition we want to investigate the composition of partially exact approximated functions w.r.t. given error bounds.

Acknowledgments. This work was supported by the German Research Foundation in the project MANIAC (DFG) (DR 287/29-1) and by the German Federal Ministry of Education and Research (BMBF) in the project EffektiV (01IS13022E).

References

[1] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *TC*, 45(9):993–1002, 1996.

[2] M. A. Breuer. Hardware that produces bounded rather than exact results. In *DAC*, pages 871–876, 2010.

[3] R. E. Bryant. Binary decision diagrams and beyond: enabling technologies for formal verification. In *ICCAD*, pages 236–243.

[4] S. T. Chakradhar and A. Raghunathan. Best-effort computing: re-thinking parallel software and hardware. In *DAC*, pages 865–870, 2010.

[5] R. Drechsler and D. Sieling. Binary decision diagrams in theory and practice. *STTT*, 3(2):112–136, 2001.

[6] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *ASPLOS*, pages 301–312, 2012.

[7] A. Fog. *How to optimize for the Pentium processor*, 1996. archived version at `http://web.archive.org/web/19961201174141/www.x86.org/ftp/articles/pentopt/PENTOPT.TXT`.

[8] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy. Low-power digital signal processing using approximate adders. *TCAD*, 32(1):124–137, 2013.

[9] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *ETS*, pages 1–6, 2013.

[10] M. C. Hansen, H. Yalcin, and J. P. Hayes. Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *D&T*, 16(3):72–80, 1999.

[11] L. Macchiarulo, L. Benini, and E. Macii. On-the-fly layout generation for ptl macrocells. In *DATE*, pages 546–551, 2001.

[12] A. Mukherjee and M. Marek-Sadowska. Wave steering to integrate logic and physical syntheses. *TVLSI*, 11(1):105–120, 2003.

[13] K. Ravi and F. Somenzi. High-density reachability analysis. In *ICCAD*, pages 154–158, 1995.

[14] M. Sauerhoff and I. Wegener. On the complexity of minimizing the OBDD size for incompletely specified functions. *TCAD*, 15(11):1435–1437, 1996.

[15] N. R. Shanbhag, R. A. Abdallah, R. Kumar, and D. L. Jones. Stochastic computation. In *DAC*, pages 859–864, 2010.

[16] D. Shin and S. K. Gupta. Approximate logic synthesis for error tolerant applications. In *DAC*, pages 957–960, 2010.

[17] D. Shin and S. K. Gupta. A new circuit simplification method for error tolerant applications. In *DATE*, pages 1566–1571, 2011.

[18] S. Venkataramani, A. Sabne, V. J. Kozhikkottu, K. Roy, and A. Raghunathan. SALSA: systematic logic synthesis of approximate circuits. In *DAC*, pages 796–801, 2012.

[19] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan. MACACO: modeling and analysis of circuits for approximate computing. In *ICCAD*, pages 667–673, 2011.

[20] R. Wille and R. Drechsler. BDD-based synthesis of reversible logic for large functions. In *DAC*, pages 270–275, 2009.

[21] R. Wille, O. Keszocze, C. Hopfmuller, and R. Drechsler. Reverse BDD-based synthesis for splitter-free optical circuits. In *ASP Design Automation Conf.*, pages 172–177, 2015.