Fast AIG-Based Approximate Logic Synthesis

Annika Heil and Oliver Keszocze

1 Introduction

A major objective of the technical industry is to provide its customers with small and fast devices which are simultaneously energy-efficient. System designers focus on three major aspects while designing a system: area, latency, and power consumption.

In terms of area, it is widely known to researchers in the field of digital technology that the number of transistors, which are able to fit on an integrated circuit, has risen steadily since the early 1960s, thanks to technological advance (also known as *Moore's law*) [1]. This trend of minimization has been declining and is expected to end in 2025 [2].

The question is, how to address the three aspects when "simply" minimizing the transistors will not be possible in the foreseeable future any more. It turns out that many applications, especially in the domain of digital signal processing, do not require strictly correct computations [3]. This is due to the fact that the human perception itself is not perfect. In some other situations, it might even be the case that the customer is willing to accept incorrect results in favor of having a faster, smaller, or less energy-hungry system [4].

A design paradigm known as *approximate computing* [5, 6] exploits this. The basic idea is to trade off computational accuracy for gains in nonfunctional aspects such as reduced area, smaller latency, and power reduction.

In the literature, two main approaches to introduce approximations to the design in order to achieve gains on one or multiple of the aspects mentioned above are used (see, e.g., [7]): (a) physical changes to the design including voltage over-scaling or overclocking or (b) altering the functionality. We will pursue the latter approach

A. Heil · O. Keszocze (✉)
Friedrich-Alexander Universität Erlangen-Nürnberg (FAU), Nürnberg, Germany
e-mail: annika.heil@fau.de; oliver.keszoecze@fau.de

R. Drechsler, S. Huhn (eds.), *Advanced Boolean Techniques*,
https://doi.org/10.1007/978-3-031-28916-3_2

in this work. More precisely, we will present a novel and fast *approximate logic synthesis* (ALS) technique. Our optimization goal is the circuit area. We refer the reader to [8] for a general survey on ALS techniques.

In this work we propose an ALS method that (a) aims to minimize the area of an approximated circuit, (b) specifically targets arithmetic circuits, (c) operates on (X)AIG representations of Boolean functions, and (d) has a small execution time due to a fast method of evaluating the error introduced by the approximation.

2 Related Work

Approximate logic synthesis has been performed on many different representations of Boolean functions using very different means of approximation.

Initial work has been done by using the structural information of a given circuit, e.g., by cutting the carry chain of adders or multipliers [9].

On a higher level of abstraction, researchers extended programming languages [10] and hardware description languages [11] with constructs to automatically compile/synthesize approximated systems.

Preliminary work on approximations on graph structures has mostly been done on BDDs [12–14]. In this work, we use the AIG data structure. The works that are closest to the work presented in this manuscript are [15] where the authors find cuts within an AIG that are replaced by approximations. The introduced error is bound by a miter structure that is evaluated using SAT. While the authors in [14] work on BDDs, we employ their idea of exploiting properties of the approximation operation to speed up the error metric computation process. We further also use their algorithmic approach for AIG approximation.

3 Background

3.1 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 $\mathrm{val}(f(x))$.

For a function f with $m = 1$, i.e., a Boolean function, its ON/OFF-set is denoted by ON / OFF(f), i.e.

$$\mathrm{ON}(f) := \{x \mid f(x) = 1\} \text{ and } \mathrm{OFF}(f) := \{x \mid f(x) = 0\}.$$

The size of a ON/OFF-set is denoted $\#\,\mathrm{ON}(f)/\#\,\mathrm{OFF}(f)$.

Given a Boolean function f, an approximated version of it is denoted by a hat, i.e., $\hat{f}$. Primary inputs are labeled A, B, ... and primary outputs X, Y, Z. Within this manuscript, the approximations do not alter the number of input or output variables.

The *truth density* $\mathrm{td}(f)$ of a function f is the ratio between the size of the ON-set of f and the total number of inputs, i.e.

$$\mathrm{td}(f) = \frac{1}{2^n} \cdot \#\,\mathrm{ON}(f).$$

The name stems from the fact that the truth density gives information about the probability of f being 1, i.e., true.

3.2 (XOR-)AND-Inverter Graphs

To efficiently represent Boolean functions, many representations have been presented. This work focuses on AND-Inverter Graphs (AIGs) [16] and XOR-AND-Inverter Graphs [17]. These structures are directed acyclic graphs. In both representations, nodes without incoming edges represent primary inputs, and nodes without outgoing edges represent primary outputs. For AIGs, the internal nodes represent the logical AND operation, whereas in a XAIG, the nodes can represent either the logical AND or the logical XOR operation. In both types of graphs, edges might be negated. We denote the size, i.e., the number of nodes, of an (X)AIG G by $\#G$.

Example 1 Consider the addition of the two-bit numbers $(CA), (DB) \in \mathbb{B}^2$, i.e., $(ZYX) = (CA) + (DB)$. An AIG representing the adder is shown in Fig. 1a. The nodes are AND operations, while dashed edges indicate negations. The output X is computed as

$$X = \underbrace{\neg \underbrace{(\neg B \wedge \neg A)}_{\text{Node6}} \wedge \neg \underbrace{(A \wedge B)}_{\text{Node5}}}_{\text{Node7}}. \tag{1}$$

Figure 1b shows an XAIG representing the same functionality, i.e., a two-bit adder. The gray nodes are XOR nodes. Note that the computation of X in Eq. (1) is actually an XOR operation, i.e., $X = A \oplus B$. This is reflected by the XAIG in node 7 that completely represents the computation of X. This shows that XAIGs may save nodes compared to AIGs. The AIG used the three nodes 5, 6, and 7 to represent the computation of X (see Fig. 1a).

While there is no one-to-one correspondence between the number of nodes in an (X)AIG and the resulting circuit size, the rule of thumb "less nodes lead to smaller circuits" does often hold and is used within this work.

In this work, we expect the functionality to be optimized by ALS to be given as an AIG. Hence, instead of optimizing a circuit, represented by an AIG, directly for

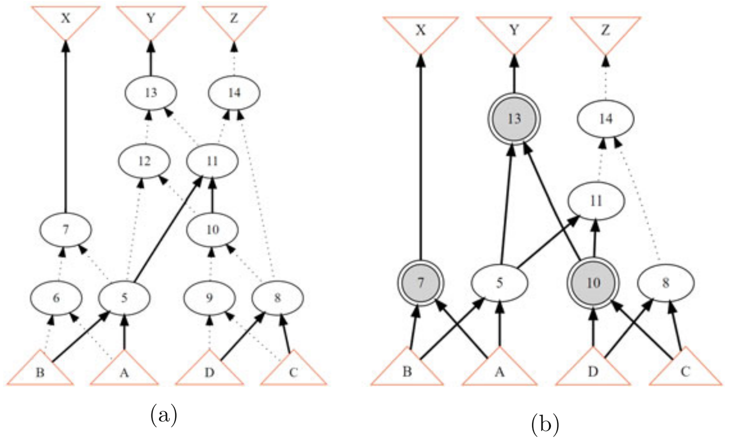


Fig. 1 AIG and XAIG for a two-bit adder. (**a**) AIG for a two-bit adder computing $(CA)+(DB) = (ZYX)$. Each node represents an AND operation. Dashed lines indicate negation. (**b**) XAIG representing the same functionality as the AIG in (**a**). The gray nodes are XOR nodes; the other nodes are AND nodes

the area used by an actual physical realization, we aim to minimize the number of AIG nodes instead.

3.3 Error Metrics

To evaluate systems in terms of the quality of the computed values, many different error metrics have been proposed. Each of these metrics measures different aspects of the approximated functionality (see [18] for an overview of commonly used metrics). Some examples of error metrics are

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

$$\mathrm{wce}(f, \hat{f}) = \max_{x \in \mathbb{B}^n} |\mathrm{val}(f(x)) - \mathrm{val}(\hat{f}(x))|, \text{ and} \tag{3}$$

$$\mathrm{whd}(f, \hat{f}) = \sum_{i=0}^{m-1} 2^i \sum_{x \in \mathbb{B}^n} \left(f_i(x) \oplus \hat{f}_i(x) \right). \tag{4}$$

The *error rate* (Eq. 2) counts how often the approximated function $\hat{f}$ computes an incorrect result. This metrics is not well-suited for evaluating approximations of arithmetic circuits as it does not take into account at all how severe the errors are as it completely ignores the actual function values. As this metrics is rather simple to evaluate (or compute an estimate using Monte Carlo simulations), it is often used

Table 1 Error metric values for the error rate, the worst-case error, and the weighted Hamming distance for an exemplary function f and its approximated function $\hat{f}$

x	$f(x)$	$\hat{f}(x)$	$f(x) \neq \hat{f}(x)$	$\lvert\mathrm{val}(f(x)) - \mathrm{val}(\hat{f}(x))\rvert$	$\sum_{i=0}^{m-1} 2^i \cdot \left(f(x) \oplus \hat{f}(x)\right)$
000	00	10	1	2	2
001	10	10	0	0	0
010	10	10	0	0	0
011	00	00	0	0	0
100	01	10	1	1	3
101	11	00	1	3	3
110	11	11	0	0	0
111	01	01	0	0	0
			$\mathrm{er}(f, \hat{f}) = 3/8$	$\mathrm{wce}(f, \hat{f}) = 3$	$\mathrm{whd}(f, \hat{f}) = 8$

in the literature. The *worst-case error* (Eq. 3) does take the values of f and $\hat{f}$ into account and returns the largest error. The last error metric (Eq. 4) is a weighted variant of the Hamming distance metric derived from the mean Hamming distance as presented in [19]. The weight parameters 2^i ensure that the bit position of an error is taken into account. Therefore, we have that errors in the more significant bits have a larger influence on the error than the lower significant bits. The metrics (3) and (4) are well-suited for arithmetic circuits.

Example 2 Table 1 shows the truth table for a function f, an approximation $\hat{f}$ of f, and the error metric values for the three error metrics introduced above.

All these error metrics have in common that they are computationally expensive to determine [20], making iterative ALS techniques that rely on repeatedly evaluating an error metric infeasible. It is possible to accelerate the error metric computation when properties of the approximation operation on a specific data structure can be exploited [14]. In this work, we adopt the greedy bucket-based algorithm from [14] to operate on (X)AIGs and choose the weighted Hamming distance as our error metric. We will use the truth density propagation from [21] to quickly compute (an estimate of) whd (see Sect. 4.3).

4 Fast AIG Approximate Logic Synthesis

4.1 Bucket-Based Approximation Algorithm

We first describe the presented ALS technique, a bucket-based approximation algorithm, on a high level of abstraction before explaining the technical details in the following sections.

The main idea behind the algorithm is, given an AIG G, to define multiple buckets that contain approximations of G that have less nodes than G. Each bucket has an error threshold. Only approximated AIGs that have an whd error lass than the threshold are stored in the bucket. All buckets are sorted in ascending order of the threshold. The algorithm iterates over the AIGs currently stored in the buckets and tries to further approximate them without exceeding the error threshold of the last bucket. When no further approximations are possible, the algorithm terminates and returns the buckets.

The returned buckets form the Pareto front for the optimization criteria number of AIG nodes (which we use as a stand-in for the circuit's area) and the weighted Hamming distance error metric.

The algorithm is depicted in Algorithm 1. In lines 1–3 the buckets are set up. They are initialized with copies of the AIG that is to be approximated; the first bucket (having the smallest error threshold) is selected as the first AIG to be approximated. The algorithm runs as long as approximations have been performed (lines 4–20). For the current bucket, nodes and corresponding approximation operations that can be applied are found (line 6). Each of these approximations

Algorithm 1: Fast approximate AIG synthesis

```
   Input   : AIG to approximate $A$, number of buckets $n$ with corresponding thresholds
   Output  : Array $bucket$ containing the approximate AIGs
   ▷ Initialize the buckets
1  $buckets \leftarrow \langle A, \ldots A \rangle$
2  $bucket[0].changed \leftarrow true$                ▷ Ensure to approximate at least once
3  $currBucket \leftarrow bucket[0]$

4  while $currBucket.changed$ do
5      $currBucket.changed \leftarrow false$
6      $approxCandidates \leftarrow findApproximationCandidates(currBucket)$; foreach
       Candidate $c \in approxCandidates$ do
7          $approx \leftarrow approximate(currBucket, c)$
8          $e \leftarrow error(approximated, A)$

           ▷ Find bucket $repBucket$ with
             • $error(approx, A) \leq error(repBucket, A)$ and
             • $\#approx < \#repBucket$.
9          $repBucket \leftarrow findFittingBucket(approxB, buckets, A)$
10         if $repBucket$ exists then
11             $repBucket \leftarrow approx$
12             $repBucket.changed \leftarrow true$
13             if $repBucket$ has a lower error threshold than $currBucket$ then
14                 $currBucket \leftarrow repBucket$              ▷ Continue with $repBucket$
15
16             else
                   ▷ Continue with next bucket
17                 $currBucket \leftarrow next(currBucket)$
18         else
19             $currBucket \leftarrow next(currBucket)$              ▷ Continue with next bucket
20

21 return $buckets$
```

is applied (line 7), and the result is evaluated for whether it can be put into one of the buckets, i.e., whether there is a bucket containing an AIG with a larger error and more nodes (line 9). If that is the case, the corresponding bucket is updated (lines 11–12). If the updated bucket has a lower error threshold than the currently used bucket, this bucket is used in the next iteration (lines 13–14); otherwise, the next bucket is used (lines 17 and 19).

We implemented the proposed ALS method in the state-of-the-art logic synthesis tool ABC [22].

4.2 Approximation Operations

In this work, we make use of the two different approximation operations, *XOR replacement* and *constant replacement*, as they can be efficiently implemented on the AIG data structure. After a replacement has been conducted, the structure of the AIG has changed, and new optimization rules may apply. Therefore, after each replacement, the AIG is again optimized by ABC.

Example 3 After replacing an input A of an AND node v (i.e., v represents $A \wedge B$) with a constant 0, e.g., allows to further replace the node v with the constant 0 as we have

$$A \wedge B \overset{\text{replace A with 0}}{\rightsquigarrow} = 0 \wedge B = 0.$$

XOR Replacement The idea behind XOR replacement is to first identify nodes in the initial AIG G that form an XOR operation and then to replace them by a single node only.

In order to identify the nodes forming an XOR operation, the AIG G is transformed into an equivalent XAIG G' (see step (a) in Fig. 2). This step is handled automatically by ABC. Note that the transformation does not necessarily replace *all* AND nodes by XOR nodes.

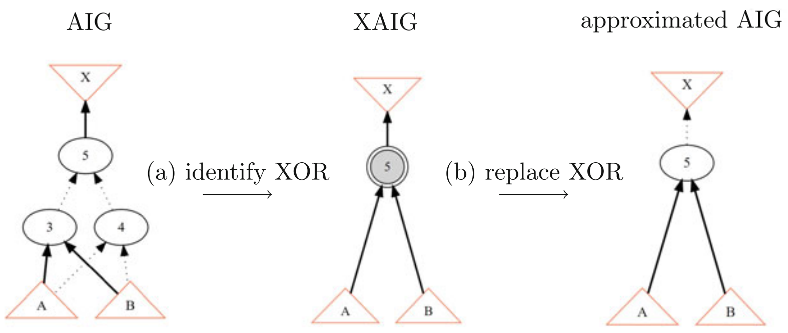


Fig. 2 Exemplary XOR replacement example

Table 2 XOR Replacements based on the truth densities of A and B

$A \oplus B$		td(A)		
		25%	50%	75%
td(B)	25%	$A \vee B$	$A \vee B$	$\neg(A \wedge B)$ / $A \vee B$
	50%	$A \vee B$	$\neg(A \wedge B)$ / $A \vee B$	$\neg(A \wedge B)$
	75%	$\neg(A \wedge B)$ / $A \vee B$	$\neg(A \wedge B)$	$\neg(A \wedge B)$

The second step (see step (b) in Fig. 2) then replaces the found XOR node by a single AND node. Note that one or multiple edges in the graph might be negated in this process (see the outgoing edge of node 5 on the right of Fig. 2).

In order to find suitable replacements for the XOR node, we investigated the XOR behavior depending on the truth densities of the inputs of the XOR operation. Table 2 shows the replacements introducing the smallest error. We obtained the replacements via exhaustive testing.

The tie breaker in the case when both NAND and OR are suitable replacements, we chose the NAND replacement when either $\text{td}(A) > (1 - \text{td}(B))$ or $\text{td}(B) > (1 - \text{td}(A))$ holds. We replace the XOR node with an OR node otherwise.

Example 4 Consider the AIG on the left of Fig. 2 and assume $\text{td}(A) = 0.7$ and $\text{td}(B) = 0.5$. The nodes 3, 4, and 5 form an XOR operation and, hence, can be replaced according to Table 2. As both NAND and OR are valid replacements, we have to check the tie breaker to decide on the actual replacement. As we have $\text{td}(B) = 0.5 > 0.3 = (1 - 0.7) = (1 - \text{td}(A))$, the three nodes are replaced by a single NAND node.

Replacing any XOR node v in the AIG of a function f according to Table 2 yields an approximation $\hat{f}$ where

$$\text{ON}(f_v) \subseteq \text{ON}(\hat{f}_v) \vee \text{ON}(\hat{f}_v) \subseteq \text{ON}(f_v) \tag{5}$$

holds. Here $f_v/\hat{f}_v$ is the function represented by the node v. Equation 5 describes over-/underapproximations, respectively. Note that the property in Eq. (5) holds only locally at the replaced node.

Constant Replacement When a node in the AIG has a truth density close to either 0 or 1, it can be considered a constant 0 or 1 node. To make this decision, the user can specify a corresponding decision threshold. As long as this threshold is less than 0.5, i.e., replace the node v with a constant $0/1$ when $\text{td}(v) < 0.5/\,\text{td}(v) > 0.5$, the constant replacement operation also has the property in Eq. (5).

For this replacement operation, the AIG G does not need to be transformed into an XAIG.

4.3 Fast Computation of the Weighted Hamming Distance

We review the definition of the weighted Hamming distance error metric from Eq. (4)

$$\text{whd}(f, \hat{f}) = \sum_{i=0}^{m-1} 2^i \cdot \left(\sum_{x \in \mathbb{B}^n} \left(f_i(x) \oplus \hat{f}_i(x) \right) \right) \tag{6}$$

and note that the computation of the Hamming distance on the individual output functions f_i can be computed using the truth density as follows:

$$= \sum_{i=0}^{m-1} 2^i \cdot \left(2^n \cdot |\text{td}(f_i) - \text{td}(\hat{f}_i)| \right) = 2^n \cdot \sum_{i=0}^{m-1} 2^i \cdot |\text{td}(f_i) - \text{td}(\hat{f}_i)|. \tag{7}$$

For this equality to hold, the function $\hat{f}$ must have been obtained by applying an approximation operation for which the property in Eq. (5) holds.

Example 5 Consider the two approximations $\hat{\text{X}}$ and $\tilde{\text{X}}$ shown in the truth table in Table 3. For the approximation operation yielding $\hat{\text{X}}$, property (5) holds, i.e., we have that $\text{ON(X)} \subset \text{ON}(\hat{\text{X}})$ holds. This property does not hold for the approximation $\tilde{\text{X}}$. Computing the whd using Eq. (7) shows that the over-/underapproximation property is crucial:

$$\text{whd}(\text{X}, \hat{\text{X}}) = 2^2 \cdot |0.50 - 0.75| = 4 \cdot 0.25 = 1$$

$$\text{whd}(\text{X}, \tilde{\text{X}}) = 2^2 \cdot |0.50 - 0.50| = 4 \cdot 0.00 = 0$$

The value $\text{whd}(\text{X}, \tilde{\text{X}}) = 0$ is clearly incorrect.

The advantage of computing whd using Eq. (7) instead of using the initial definition of Eq. (4) is that the actual time necessary to determine the value can be greatly reduced if the computation of the truth densities can be done quickly. We will see in Sect. 4.4 how this is possible.

Table 3 Approximating *X* using operations for which the over-/underapproximation property Eq. (5) does hold ($\hat{\text{X}}$) and does not hold ($\tilde{\text{X}}$)

A	B	X	$\hat{\text{X}}$	$\tilde{\text{X}}$
1	1	0	0	0
0	1	1	1	1
1	0	1	1	0
0	0	0	1	1

As the proposed ALS method (see Algorithm 1) is an iterative approach, many whd values have to be computed during a synthesis run. When the total number of inputs does not exceed 16, the AIG can be fully evaluated and exact results can be computed. When the AIG grows beyond this, the truth density and, hence, the whd are computed iteratively by determining the whd locally for the approximated node only. We then adopt an additive model accumulating the locally computed errors until the output node is reached. This additive model along with the fact that consecutive errors that might cancel each other out (a situation also known as *error masking*) are not taken into account leads to an overestimation of the total error. The upside of this simplification is that it allows for a very fast estimation of the total whd.

4.4 Truth Density Computation

So far, we used the truth density values of all (X)AIG nodes without considering how to actually compute them. In this work, we make use of two different means of obtaining the truth densities of the nodes.

The first means of obtaining the truth density is to directly use ABC. The tool estimates the truth density values of the nodes by running a number of simulations of the graph, i.e., evaluating the graph for a given number of randomly generated inputs. The quality of the result greatly varies with the number of simulations and, hence, the time one is willing to spend on the estimation.

As the computation of the densities is crucial for both, the decision on which node to replace and the computation of the weighted Hamming distance error metric, we chose to use the error propagation method presented in [21]. While its intended use is to propagate the error rate through a general Boolean network, it can easily be applied for our use case as (X)AIGs are nothing but a specific Boolean network and the truth density is already computed by the approach as a "by-product."

The speed of the approach from [21] stems from not having to perform full simulations of the (X)AIGs but computes the truth density using symbolic variables. It should be noted that computed densities are only exact in case when there is no fanout reconvergence in the (X)AIG. Nevertheless, extensive tests have shown that the degradation of the results in case of reconvergences is negligible.

5 Experimental Evaluation

5.1 Experimental Setup

We implemented the proposed ALS technique in the state-of-the-art logic synthesis tool ABC [22] using the probabilistic error propagation tool from [21].

As benchmark circuits, generic n-bit adders and multipliers as well as the EvoApproxLib$^{\text{LITE}}$ library [18] are used.

Instead of directly specifying the whd value of the buckets, we define a *threshold* $t \in [0, 1]$ that reflects how large the error in the most significant bit of the output is allowed to be in percent. This allows to define buckets that capture similar error behavior for circuits of different size, i.e., one does not have to (manually) compute different bucket values for an 8-bit and an 16-bit multiplier. A threshold value t can be translated in an estimate on the whd via $\text{whd}(f, \hat{f}) \approx t \cdot 2^n \cdot 2^{m-1}$.

All experiments were executed on an AMD Ryzen 5 3600XT 6-Core CPU with 3.80-GHz and 16-GB memory running Ubuntu 20.04 in WSL 2 on Windows 10 Build 19044.1645.

5.2 Scalability

To assess the scalability of our approach, we performed approximate logic synthesis on adders of increasing bit width using 5 buckets with threshold values 0.0156, 0.0.03125, 0.0625, 0.125, and 0.25. In the experiments, it turned out that the error propagation implementation has a memory leak preventing it to be used for AIGs with more than $\approx$ 300 nodes. Therefore, the following results were obtained using ABC's simulation method.

The results of the synthesis runs are presented in Table 4. For each bit width, the results for each bucket are listed in a separate line. The approximated AIGs were converted to a list of logic gates using ABC. Afterward, the area and delay have been computed by ABC using the `mcnc.genlib` gate library. For each physical aspect, the number of gates, the area, and the delay, we present the reduction/increase in the aspect in percent after the absolute values in the table. We further report the whd for the AIGs.

Using the number of AIG nodes as a stand-in for the circuit area works well: the reduction in nodes is qualitatively reflected in the reduction in the number of gates and the reduction of the area. As can be seen, the goal of optimizing circuits for area has been achieved. It is interesting to see the reduction remains in the range $\approx 65\%$–75% for threshold values up to 0.125. Only after allowing for 25% weighted errors in the most significant bit, further size reductions are achieved.

While our method is capable of reducing the area, it does, in turn, increase the delay of the circuit (usually in the $\approx 112\%$–125% range). This value, again, drops when a large threshold is used. As we do not explicitly optimize for delay, this is an acceptable trade-off.

As can be seen, the actual whd values for the buckets increase with increasing bit width of the adders. This shows that choosing a means to describe buckets that abstracts away the bit width is helpful.

Table 4 ALS results for adders of varying bit width. For each bucket, the number AIG node, number of gates, area, delay, and whd are reported. For the number of nodes/gates, the area, and the delay, the relative change to the unapproximated AIG is also shown

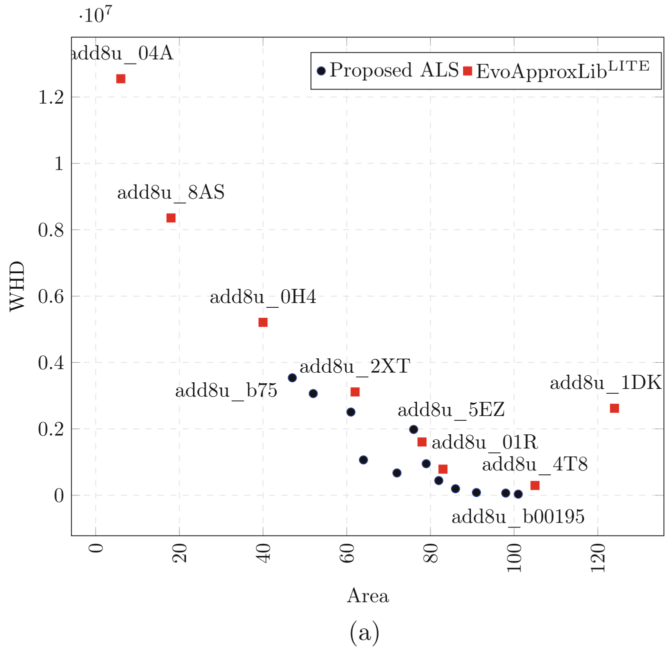
Threshold	Nodes	%	Gates	%	Area	%	Delay	%	whd
16 bit [Run-time: 34.64 s]									
Unapprox.	158	–	105	–	215	–	33.70	–	–
0.0156	141	89%	66	63%	170	79%	39.30	117%	2.75
0.03125	139	88%	66	63%	166	77%	38.80	115%	3.00
0.0625	137	87%	66	63%	162	65%	38.30	114%	3.25
0.125	135	85%	66	63%	158	73%	37.80	112%	3.50
0.25	131	83%	61	58%	151	70%	35.30	104%	4.00
32 bit [Run-time: 365.54 s ≈ 5 m]									
Unapprox.	318		217	–	439	–	65.70	–	–
0.0156	270	85%	131	60%	319	73%	77.40	118%	7.00
0.03125	269	85%	132	61%	315	72%	77.20	118%	7.25
0.0625	266	84%	133	61%	309	70%	78.00	119%	7.75
0.125	264	83%	132	61%	306	70%	77.70	118%	8.00
0.25	193	60%	83	38%	176	40%	44.60	66%	16.64
64 bit [Run-time: 2527.54 s ≈ 42 m]									
Unapprox.	638	–	441	–	887	–	129.70	–	–
0.0156	523	82%	261	59%	598	67%	160.30	124%	15.50
0.03125	521	82%	261	59%	594	67%	159.80	123%	15.75
0.0625	519	81%	261	59%	590	67%	159.30	123%	16.00
0.125	517	81%	261	59%	586	66%	158.80	122%	16.25
0.25	461	72%	223	51%	483	54%	134.40	103%	21.45
128 bit [Run-time: 22738.28 s ≈ 6 h]									
Unapprox.	1278	–	889	–	1783	–	257.70	–	–
0.0156	1034	81%	516	58%	1179	66%	315.70	123%	32.48
0.03125	1032	81%	516	58%	1175	66%	315.20	123%	32.73
0.0625	1030	81%	516	58%	1171	66%	314.70	122%	32.98
0.125	1028	80%	516	58%	1167	65%	314.20	122%	33.23
0.25	501	39%	194	22%	211	12%	57.20	22%	61.72

5.3 *Multi-Objective Optimization for Area and* whd

The benchmark library EvoApproxLib$^{\text{LITE}}$[1] [18, 23] provides a selection of approximate adders and multipliers. They have been synthesized via exhaustive search with respect to various error metrics (including er and wce) as well as area and power consumption. The benchmark set does not evaluate the whd error metric.

As the final buckets of the presented approach form the Pareto front of the multi-objective optimization problem with the optimization criteria whd and area, we

[1] The benchmark library is publicly available at https://ehw.fit.vutbr.cz/evoapproxlib/.



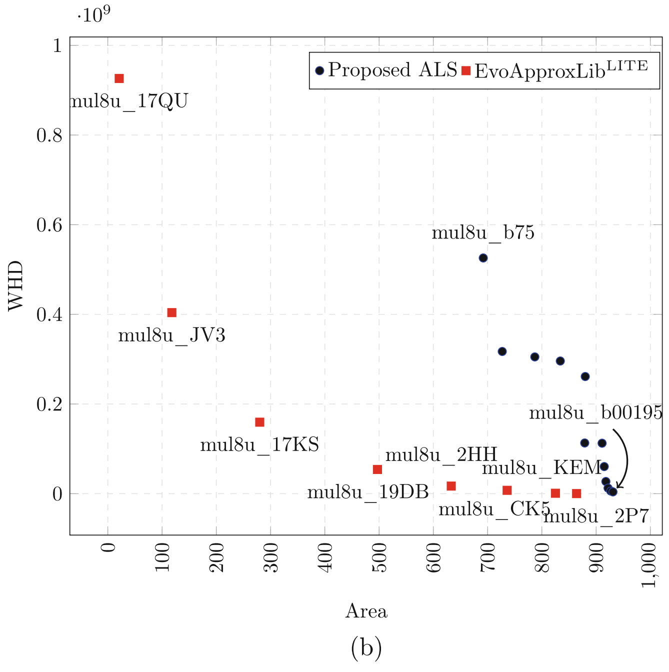


Fig. 3 Comparison of the synthesis results of the proposed approach (dark blue dots) and EvoApproxLib$^{\text{LITE}}$ (red squares) with respect to area and the weighted Hamming distance error metric for (**a**) 8-bit unsigned adders and (**b**) 8-bit unsigned multipliers

compare our results with the exhaustive results from EvoApproxLib$^{\text{LITE}}$. We select the adders and multipliers from the benchmark set that form the Pareto front with respect to area and the wce error metric and compute the whd values for them so that we can compare the benchmark circuits to our results. The circuits optimized for this metric were chosen as wce does take into account the order of the output bits, and, therefore, the corresponding circuits allow for the fairest comparison.

The comparison for 8-bit unsigned adders and multipliers is shown in Fig. 3. The notation for our circuits is as follows: "add8u_b75" refers to an unsigned 8-bit adder from the bucket with a threshold of 0.75. For EvoApproxLib$^{\text{LITE}}$, the naming scheme is of the form "add8u_⟨ID⟩" and directly taken from their website.

For the adders (Fig. 3a), the proposed ALS method clearly produces better results than EvoApproxLib$^{\text{LITE}}$. These results can be explained, in part, by the fact that EvoApproxLib$^{\text{LITE}}$ optimized for a different error and in part by the fact that the computation of the sum bits in an adder basically is a large XOR gate. When looking at the results for the multipliers (Fig. 3b), one can see that the applied approximations are not resulting in points close to the Pareto front any more. When investigating what approximation operations have been chosen by the proposed ALS algorithm (see Table 5), one can see that the ratio of XOR replacement over constant replacements for the adder is higher than for the multiplier. This further hints that XOR replacement is well-suited for adders while multipliers do not benefit from this particular kind of approximation.

5.4 *Truth Density Computation*

To investigate the difference in execution time between the ABC simulation-based truth density estimation and the method from [21], we synthesized adders of increasing bit width using 5 buckets with threshold values 0.0156, 0.0.03125, 0.0625, 0.125, and 0.25. The results are reported in Fig. 4. As can be seen, the error propagation approach clearly excels with respect to the execution time. Due to the memory leakage issue (see Sect. 5.2), we can not show results for larger circuits. While using ABC for the error estimation already is fast, using error propagation shows a great potential to further accelerate our proposed ALS technique.

Table 5 Number of XOR/Constant replacements for 8-bit adders and multipliers

	Replace	Replace	Ratio
Circuit	XOR	Constant	
8-bit adder	37	4	9.25
8-bit multiplier	136	38	3.6

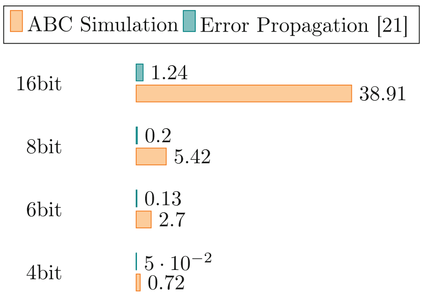


Fig. 4 Execution time of the proposed ALS method in seconds for adders of varying bit width

6 Conclusion and Outlook

We presented a novel and fast greedy, bucket-based approximate logic synthesis technique working on AIGs that aims to minimize both the area of the resulting circuit and, at the same time, the error introduced by the approximations. We chose the weighted Hamming distance error metric whd to assess the functional quality of the circuit as it takes into account the order of the output bits. We found a means of effectively computing whd via computing truth densities and exploiting properties of the used approximation operations. The effectiveness of the presented method has been evaluated in a set of experiments.

The next step is to find the memory leakage in the fast error propagation tool to further enhance the speed of the proposed method.

Acknowledgments The paper has been partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—450987171.

References

1. Moore, G.E.: Cramming more components onto integrated circuits. Electronics **38**(8), pp. 114 (1965)
2. Waldrop, M.M.: The chips are down for Moore's law. Nature News **530**(7589), 144–147 (2016). Visited on 06/20/2022
3. Zhu, N., et al.: Design of low-power high-speed truncation-error-tolerant adder and its application in digital signal processing. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **18**(8), 1225–1229 (2010)
4. Venkatesan, R., et al.: MACACO: modeling and analysis of circuits for approximate computing. In: International Conference On Computer Aided Design (2011). Visited on 09/19/2018
5. Han, J., Orshansky, M.: Approximate computing: an emerging paradigm for energy-efficient design. In: European Test Symposium (2013)
6. Mittal, S.: A survey of techniques for approximate computing. ACM Comput. Surv. **48**(4), 1–33 (2016). Visited on 02/28/2019
7. Venkataramani, S., et al.: Approximate computing and the quest for computing efficiency. In: Design Automation Conference (2015). Visited on 04/11/2019

8. Scarabottolo, I., et al.: Approximate logic synthesis: a survey. Proc. IEEE **108**(12), 2195–2213 (2020)
9. Shafique, M., et al.: A low latency generic accuracy configurable adder. In: Design Automation Conference (2015). Visited on 02/08/2019
10. Barbareschi, M., Iannucci, F., Mazzeo, A.: A pruning technique for B&B based design exploration of approximate computing variants. In: International Symposium on VLSI (2016)
11. Keszocze, O., Kiessling, M.: Approximate computing extensions for the clash HDL compiler. In: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (2021)
12. Soeken, M., et al.: BDD minimization for approximate computing. In: Asia and South Pacific Design Automation Conference (2016)
13. Shirinzadeh, S., et al.: An adaptive prioritized e-preferred evolutionary algorithm for approximate BDD optimization. In: Proceedings of the Genetic and Evolutionary Computation Conference (2017). Visited on 10/19/2018
14. Keszocze, O.: BDD-based error metric analysis, computation and optimization. IEEE Access **10**, 14013–14028 (2022)
15. Chandrasekharan, A., et al.: Approximation-aware rewriting of AIGs for error tolerant applications. In: International Conference On Computer Aided Design (2016). Visited on 10/24/2018
16. Mishchenko, A., Chatterjee, S., Brayton, R.: DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In: Design Automation Conference (2006). Visited on 07/31/2019
17. Háleček, I., Fišer, P., Schmidt, J.: On XAIG rewriting. In: International Workshop on Logic & Synthesis (2017)
18. Mrazek, V., 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). Visited on 10/17/2018
19. Vasicek, Z.: Formal methods for exact analysis of approximate circuits. IEEE Access **7**, 177309–177331 (2019)
20. Keszocze, O., Soeken, M., Drechsler, R.: The complexity of error metrics. Inf. Process. Lett. **139**, 1–7 (2018). Visited on 08/08/2018
21. Echavarria, J., et al.: Probabilistic error propagation through approximated Boolean networks. In: Design Automation Conference (2020)
22. Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification. University of California, Berkeley (2010)
23. Mrazek, V., Vasicek, Z., Sekanina, L.: EvoApproxLib: extended library of approximate arithmetic circuits. In: Workshop on Open-Source EDA Technology (2019)