Approximation-aware Rewriting of AIGs for Error Tolerant Applications

Arun Chandrasekharan[1] Mathias Soeken[2] Daniel Große[1,3] Rolf Drechsler[1,3]
[1]Group of Computer Architecture, University of Bremen, Germany
[2]Integrated Systems Laboratory, EPFL, Switzerland
[3]Cyber Physical Systems, DFKI GmbH, Bremen, Germany
{arun,grosse,drechsle}@cs.uni-bremen.de, mathias.soeken@epfl.ch

ABSTRACT

Approximation circuits offer superior performance (speed and area) compared to traditional circuits at the cost of computational accuracy. The accuracy of the results in approximation circuits is evaluated based on several error metrics such as worst-case error, bit-flip error, or error-rate. Several applications have varied requirements in error metrics, i.e., all the error criteria have to be met together at a time, or in combinations. Nevertheless, all applications benefit from improved delay and area. An automated synthesis approach with formal guarantees on error metrics is very helpful in generating circuits that meet these criteria. Furthermore, each of these metrics are independent quantities (value of one metric does not correlate with the other), and automated synthesis can discover opportunities to trade off one or more of the relaxed metrics with a strict requirement on the other, resulting in better performance.

In this paper, we present an automatic synthesis approach using And-Inverter Graphs (AIGs) based rewriting that not only improves the performance but also guarantees the bounds of approximation errors introduced. Our synthesis approach is evaluated on a wide range of designs and standard benchmark circuits to show the usefulness and applicability. In particular, we show that our synthesis results are even comparable with the optimization achieved with hand crafted adhoc approximation circuits such as approximation adders in a case study on image compression.

1. INTRODUCTION

Approximate computing improves the efficiency of a circuit in terms of speed and area by relaxing computational accuracy. At a first glance one might think that this approach is not a good idea, but it has become evident that there is a huge set of applications which can tolerate errors. Applications such as multi-media processing and compressing, voice recognition, web search, or deep learning are just a few examples.

ICCAD '16, November 07-10, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4466-1/16/11...$15.00
DOI: http://dx.doi.org/10.1145/2966986.2967003

Certainly, several questions arise when rethinking the circuit design process under the concept of approximate computing: (1) What errors are acceptable for a concrete application? (2) How to design approximate circuits? (3) How to perform functional verification and production test? All of these are major questions. For answering the first question, different error metrics have been proposed. Essentially, they measure the approximation error by comparing the output of the original circuit against the output of the approximation circuit. Typical metrics are error-rate, worst-case error, and bit-flip error [3]. Please note that the chosen metric depends highly on the application.

On the design side (second question), we focus on *functional approximation* in this work, i.e., a slightly different function is realized (in comparison to the original one) resulting in a more efficient implementation. Two main directions of functional approximation can be distinguished: (i) for a given design, an approximation circuit is created manually; most of the research has been done here. This includes for example approximate adders [8, 9] and approximate multipliers [10]. However, since this procedure has strong limitations in making the potential of approximation widely available, research started on (ii) design automation methods to derive approximated components from a golden design automatically.

Different synthesis approaches have been proposed. They range from reduction of sum-of-product implementations [22], redundancy propagation [23], and don't care based simplification (SALSA) [26] to dedicated three-level circuit construction heuristics [2]. Recently the ASLAN [20] framework has been presented which extends SALSA and is able to synthesize approximate sequential circuits. ASLAN uses formal verification techniques to ensure quality constraints given in the form of a user-specified *Quality Evaluation Circuit* (QEC). However, the QEC has to be constructed by the user similar to a test bench, which is a design problem by itself. In addition, constructing a circuit to formulate the approximation error metrics requires detailed understanding of formal property checking (liveness and safety properties), and verification techniques. Further, some error metrics such as error-rate cannot be expressed in terms of Boolean functions efficiently since these require counting in the solution space which is a #SAT problem (i.e., counting the number of solutions). Moreover the error metrics used in these approaches are rather restricted (e.g., [26] uses worst-case error and a very closely related relative-error as metrics) and how to trade off a stricter requirement in one metric wrt. to a relaxed requirement in another has not been considered, especially when the error metrics are unrelated to each other.

In this paper, we propose an algorithm for the synthesis of approximation circuit with formal guarantees on error metrics. We introduce approximation-aware AIG rewriting and our approach is able to synthesize circuits with significantly improved performance within the allowed error bounds. Experimental evaluation of our approach on a broad range of applications confirm the applicability of our method. We also compare the quality to manually handcrafted approximate designs. Further, the approximation-aware rewriting technique can trade-off the relative significance of each error metric for a particular application, to improve the quality of the synthesized circuits.

In short, the major contributions in this paper are as follows.

1. We propose a synthesis methodology based on AIG rewriting for approximate computing with formal guarantees on errors.

2. Our approach provides the ability to trade off one error criteria with another.

3. The results show comparable quality to manually handcrafted approximate designs.

The remainder of this paper is structured as follows. Section 2 provides the necessary preliminaries. Section 3 gives an overview of our approximation-aware AIG rewriting approach. We explain the implementation details of our methodology in Section 4 and describe the results of the experimental evaluation in Section 5. Finally, the paper is concluded in Section 6.

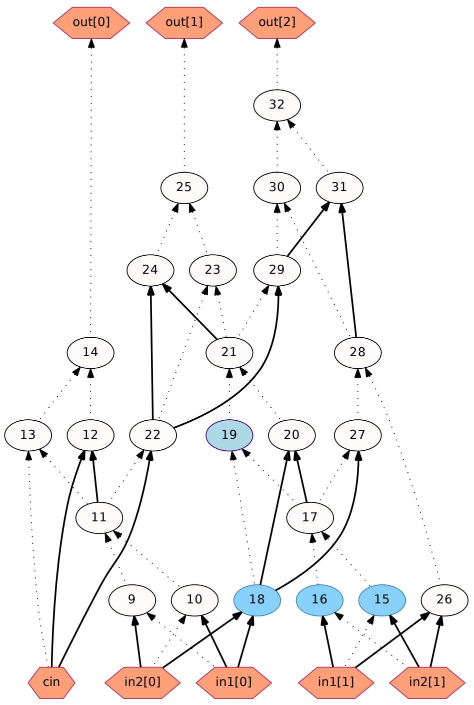


Figure 1: AIG for 2-bit Full Adder

2. PRELIMINARIES

2.1 And-Inverter Graph

A *Boolean network* is a directed acyclic graph (DAG) where nodes represent logic gates or primary inputs/primary outputs (PIs/POs), and edges represent wires that form the interconnection among the gates. An *And-Invertor Graph* (AIG) is a Boolean network where the logic gates are two-input ANDs and the edges can be complemented, i.e., inverted.

A *path* in an AIG, is a set of nodes starting from a PI or a constant, and ending at a PO. The *length* of the path is the number of nodes in the path excluding the PI and PO. The *depth* of an AIG is the maximum length among all the paths and the *size* is the total number of nodes in the AIG. The depth of an AIG corresponds to delay and size corresponds to area. It must be noted that the area and the delay of the final implemented circuit heavily depends on the technology mapping, routing delay and other technological parameters and as such cannot be deduced from synthesis itself. However in the non-mapped network, depth and size of AIG graph correlates well with delay and area of the circuit.

Rewriting is an algorithmic transformation in an AIG that introduces local modifications to the network to reduce the depth and, or the size of the AIG [18]. Rewriting takes a greedy approach by iteratively selecting subgraphs rooted at a node and substituting them with better pre-computed subgraphs.

A *cut* of a node v is a set of nodes C, called *leaves*, such that (i) every path from v to a primary input must visit at least one node in C, and (ii) every node in C must be included in at least one of these paths [5]. The node v is called the *root* of the cut, and there may be several cuts for one node. A cut is called *k-feasible*, if $|C| \leq k$. The *cut size* is the number of nodes in the transitive fan-in cone of v up to the leaves without including them. Each cut expresses a *cut function* at the root expressed in terms of its leaves as inputs. Intuitively a k-feasible cut represent a single output function g, with k inputs which may be shared or substituted with another function, $\hat{g}$. For rule based synthesis rewriting, the substituted function is an *equivalent* function conforming to the desired synthesis goals [18, 6, 11]. However, in approximate synthesis, the substituted function does not have to be equivalent, but should respect the global error and quality metrics, and synthesis goals. Cuts in an AIG can be computed using *cut enumeration* techniques [19].

AIGs can be used to efficiently represent both sequential and combinational circuits. In this work we restrict ourselves to combinational networks only. An AIG network of a 2-bit *full adder* circuit is illustrated in Fig 1. Each node other than the terminal nodes (PIs and POs) represent an AND gate, and the dotted arrows indicate inversion of the respective input. A 3-input cut is shown with root node *19* and leaves *18, 16* and *15*. The size of this cut is 2 (nodes *19*, *17*).

2.2 Combinational Equivalence Checking

Combinational equivalence checking (CEC) refers to the usage of formal techniques to exhaustively verify two different implementations of the same logic to be functionally *equivalent* i.e. both the implementations produce the same output under all possible input combinations [14]. CEC when applied to approximate hardware aims to establish whether both the implementations produce outputs that do not differ beyond a predefined quality threshold, under all possible input combinations.

For a Boolean function $f(x)$, *Boolean Satisfiability* (SAT) problem consists of finding a satisfying input assignment where f evaluates to *true*, or absence of such assignment. SAT is a decision problem and finding any one solution returns *true*. SAT techniques have developed tremendously over the past few decades and the current state-of-the-art SAT solvers can solve millions of clauses in moderate time [25]. In order to show that two implementations are equivalent, a *miter* circuit is created and the output of the miter is proved to be always $false$ using SAT.

2.3 Error Metrics

There are several error metrics proposed to evaluate the quality of approximate synthesis [3, 27, 15]. The error metrics relevant to our work are briefly described below.

Let $f : \mathbb{B}^n \to \mathbb{B}^m$ be a Boolean function and $\hat{f} : \mathbb{B}^n \to \mathbb{B}^m$ be the approximated version of it. Then, the *worst-case error* is defined as the maximum among the absolute differences of f and $\hat{f}$ over all the possible input combinations. The worst-case error[1] is given by the formula

$$e_{\mathrm{wc}}(f,\hat{f}) = \max_{x\in\mathbb{B}^n} \left|\mathrm{int}(f(x)) - \mathrm{int}(\hat{f}(x))\right|, \tag{1}$$

where 'int' denotes the integer representation of the bit vector.

The maximum *bit-flip error*, e_{bf}, is defined as the maximum hamming distance between f and $\hat{f}$.

$$e_{\mathrm{bf}}(f,\hat{f}) = \max_{x\in\mathbb{B}^n} \left(\sum_{i=0}^{m-1} \left| f_i(x) - \hat{f}_i(x) \right| \right), \tag{2}$$

where m is the width of the output bit vector.

Finally, the *error-rate*

$$e_{\mathrm{er}}(f,\hat{f}) = \frac{\sum\limits_{x\in\mathbb{B}^n} [f(x) \neq \hat{f}(x)]}{2^n} \tag{3}$$

is the fraction of input assignments that lead to a different output pattern out of total number of input patterns.

The bit-flip error and error-rate are not related to the magnitude of the approximated output. In particular, error-rate involves counting the correct solutions and the value of error itself is immaterial. Furthermore all these error metrics are independent quantities on their own and do not necessarily correlate with each other. For e.g., a design with very high worst-case error does not imply that the bit-flip error or the error-rate is high.

3. OVERVIEW

Our approach applies network rewriting which allows to change the functionality of the circuit, but does not allow to violate given error behavior. The error behavior is given in terms of thresholds to error metrics. It is possible that a combination of several error metrics is given.

[1]also referred to as *error-significance*

Algorithm 1 Approximation Rewriting

```
 1: function APPROX_REWRITE(AIG $f$, error behavior $e$)
 2:     set $\hat{f} \leftarrow f$
 3:     while continue do
 4:         set paths $\leftarrow$ select_paths($\hat{f}$)
 5:         for each $p \in$ paths do
 6:             set cuts $\leftarrow$ select_cuts($p$)
 7:             for each $C \in$ cuts do
 8:                 set $\hat{f}_{\mathrm{cnd}} \leftarrow$ replace $C$ by $\hat{C}$ in $\hat{f}$
 9:                 if $e(f, \hat{f}_{\mathrm{cnd}})$ then
10:                     set $\hat{f} \leftarrow \hat{f}_{\mathrm{cnd}}$
11:                 end if
12:             end for
13:         end for
14:     end while
15:     return $\hat{f}$
16: end function
```

The rewriting algorithm is outlined in Algorithm 1. The description is generic and details on the important steps are described in the next section. Input is an AIG that represents some function f. It returns an AIG that represents an approximated function $\hat{f}$, which complies to the given error behavior e. In the algorithm we model the error behavior as a function that takes as input f and $\hat{f}$ and returns 0, if the error behavior is violated. As an example, we can define the error behavior

$$e(f,\hat{f}) = e_{\mathrm{wc}}(f,\hat{f}) \leq 1000 \wedge e_{\mathrm{bf}}(f,\hat{f}) \leq 5$$

in which the worst-case error should be less than 1000 and the maximum bit-flip error should be less than 5.

The algorithm initially sets $\hat{f}$ to f (line 2). It then selects paths in the circuit to which rewriting should be applied (line 4). Cuts are selected from the nodes along these paths (line 6). For each of these cuts C, an approximation $\hat{C}$ is generated, and inserted as a replacement for C. The result of this replacement is temporarily stored in the candidate $\hat{f}_{\mathrm{cnd}}$ (line 8). It is then checked, whether this candidate respects the error behavior. If that is the case, $\hat{f}$ is replaced by the candidate $\hat{f}_{\mathrm{cnd}}$ (line 10). This process is iterated as long as there is an improvement, based on a user provided limit on number of attempts, or as long given resource limits have not been exhausted (line 3).

4. IMPLEMENTATION

In this section, we describe details on how to implement Algorithm 1. The crucial parts in the algorithm are (i) which paths are selected, (ii) which cuts are selected, (iii) how cuts are approximated, and (iv) how the error behavior is evaluated.

The approximation rewriting algorithm is an iterative approach and a decision has to be taken on how many iterations need to be run before exiting the routine. This is implemented as effort levels (*high*, *medium*, and *low*) in the tool corresponding to number of paths selected for approximation. The user has to specify this option. Alternately user can also specify the number of attempts tried by the tool.

4.1 Select Paths

The primary purpose of the proposed approximation techniques is to reduce delay and area of the circuits. In order to reduce delay, we select the critical paths, i.e., the longest paths in the circuit. Replacing cuts on these paths with approximated cuts of smaller depth reduces the overall depth of the circuit. In our current implementation, we select all critical paths. The set of critical paths changes in each iteration.

4.2 Select Cuts

While selecting critical paths potentially allows to reduce the depth of the approximated circuit, selecting cuts allows to reduce area. We select cuts by performing cut enumeration on the selected paths. In our implementation the enumerated cuts are sorted based on the increasing order of cut size. The rational for approximating the cuts based on increasing order of cut size is as follows. For a given path in the AIG, if we assume each node has equal probability in inducing errors, the size of the cut can be related to the perturbations introduced in the network and therefore the least impact. Hence starting with a transformation that introduce minimum errors has the best chance of introducing approximations without violating error metrics and falling into a local minima quickly. Although this assumption appears oversimplified, the error metrics (worst-case error, bit-flip error, and error-rate) are independent quantities and do not necessarily correlate with each other. Hence a quick and efficient way to prioritize cuts is based on cut size going along with the assumption. Our experimental results also confirm the applicability of such an approach. This is the default behavior of the tool. We have tried experiments with maximum cut size first, but this scheme is observed to be falling into local minima at a faster rate and the results are inferior. This further confirms that prioritizing cuts based on increasing order of size, per path, is the most acceptable way. Sometimes selecting cuts randomly for approximation benefits the rewriting procedure. In the current implementation of the tool, this behavior can be optionally enabled by the user.

4.3 Approximate Cut

Each cut is replaced by an approximated cut to generate a candidate for the next approximated network. Ideally, one would like to approximate the cut with a similar cut of better performance, i.e., the error of the function of the approximated cut has minimal errors wrt. to the original cut function, but maximal savings in area and delay. In our current implementation, we simply replace the cut by constant 0, i.e., the root node of the cut is replaced by the constant 0 node. This trivial routine is found to be sufficient for good overall improvements in our experimental evaluation. Investigating how to approximate cuts in a nontrivial manner is a potential area of future research to gain further improvement.

4.4 Evaluate Error

To compute whether the error behavior is respected, we need a way to precisely compute the error metrics. For this purpose, we make use of an *approximation miter* [4]. An approximation miter takes as input two networks C and $\hat{C}$, an error computation network E, and a decision circuit D. The output of the miter is a single bit *bad* which evaluates to 1 if and only if the error is violated. The general configuration of an approximation miter for combinational networks is illustrated in Figure 2.

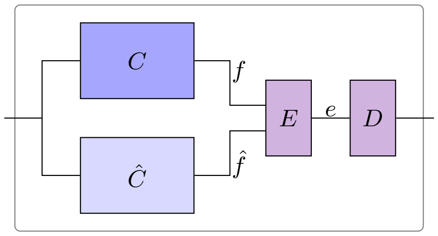


Figure 2: Approximation Miter

Algorithm 2 Finding maximum bit-flip error

```
 1: function FIND_MAX_BIT_FLIP_ERROR
 2:     lbound ← 0
 3:     ubound ← $m - 1$
 4:     while lbound < ubound do
 5:         $X \leftarrow \left\lceil \frac{(\text{ubound} + \text{lbound})}{2} \right\rceil$
 6:         $s \leftarrow \text{SAT}\left(\text{ApproxMiter}\left(\sum_{i=0}^{m-1}\left(f_i \oplus \hat{f}_i\right), X\right)\right)$
 7:         if $s = \textit{satisfiable}$ then
 8:             lbound ← $X$
 9:         else
10:             ubound ← $X - 1$
11:         end if
12:     end while
13:     return lbound
14: end function
```

The error computation network E and the decision network D can be configured to do the error analysis after applying approximation rewriting to AIG. In this work, the worst-case error and bit-flip error are evaluated using the approximation miter. The evaluation of error-rate involves the counting of solutions in $\hat{f}$ that differ from f. The algorithm given in [24] is used for error-rate computations using *Binary Decision Diagram* (BDD). The approach used is an exact one, unique to BDD based representation and so far there is no equivalent algorithm using AIG.[2] A direct consequence of this is extended run-times necessitated by a conversion from AIG to BDD.

The error metrics e_{wc} and e_{er} can be precisely computed for combinational circuits with the symbolic algorithms given in [24]. However for e_{bf}, we extend the concept outlined in [4]. It is formulated as an optimization problem using approximation miter and computed with binary search and SAT. The binary search is shown in Algorithm 2. For a function f with output width m, X is set to one half of m in the first loop, with lower bound 0 and upper bound $m-1$. SAT is used to solve the approximation miter and if SAT returns *satisfiable*, the lower bound is set to X, else the upper bound is set to $X-1$. The binary search algorithm iterates further until the bounds converge to e_{bf}.

[2]Explicit enumeration of solutions is not considered.

Table 1: Error Metrics comparison for Approximation Adders

Approximation Architecture							Approximation Synthesis							
Architecture	Gates⋆	Delay⋆ (*ns*)	Area⋆	e_{wc}	e_{er} (%)	e_{bf}	Synthesis Scheme† ($e_{\text{wc-in}}$, $e_{\text{bf-in}}$)	Gates⋆	Delay⋆ (*ns*)	Area⋆	e_{wc}	e_{er} (%)	e_{bf}	run-time (*sec*)
8-bit Adders							8-bit Adders							
RCA_N8‡	57	10.2	175	0	0.00	0	RCA_N8‡	57	10.2	175	0	0.00	0	0
ACA_II_N8_Q4 ±	39	7	137	64	18.75	5	appx1 (64, 5)	41	7	138	64	75.00	4	50
ACA_I_N8_Q5	52	7	175	128	4.69	4	appx2 (128, 4)	27	7	86	128	78.22	4	57
GDA_St_N8_M4_P2 ∓	39	7	137	64	18.75	5	appx1 (64, 5)	41	7	138	64	75.00	4	50
GDA_St_N8_M4_P4	37	9	134	64	2.34	3	appx3 (64, 3)	36	8.6	121	64	50.00	3	68
GDA_St_N8_M8_P1	26	3.8	108	168	60.16	7	appx4 (168, 7)	13	3.8	33	128	96.94	7	11
GDA_St_N8_M8_P2	35	5.4	124	144	30.08	6	appx5 (144, 6)	15	5.4	45	144	94.75	6	56
GDA_St_N8_M8_P3	45	7	149	128	12.50	5	appx6 (128, 5)	19	7	64	128	88.33	5	22
GDA_St_N8_M8_P4	44	7	157	128	4.69	4	appx2 (128, 4)	27	7	86	128	78.22	4	57
GDA_St_N8_M8_P5	63	8	194	128	1.56	3	appx7 (128, 3)	31	8.6	104	128	62.70	3	58
GeAr_N8_R1_P1 ‡‡	26	3.8	108	168	60.16	7	appx4 (168, 7)	13	3.8	33	128	96.94	7	11
GeAr_N8_R1_P2	35	5.4	124	144	30.08	6	appx5 (144, 6)	15	5.4	45	144	94.75	6	56
GeAr_N8_R1_P3	47	7	153	128	12.50	5	appx6 (128, 5)	19	7	64	128	88.33	5	22
GeAr_N8_R1_P4	52	7	175	128	4.69	4	appx2 (128, 4)	27	7	86	128	78.22	4	57
GeAr_N8_R1_P5	43	8.6	147	128	1.56	3	appx7 (128, 3)	31	8.6	104	128	62.70	3	58
GeAr_N8_R2_P2	39	7	137	64	18.75	5	appx1 (64, 5)	41	7	138	64	75.00	4	50
GeAr_N8_R2_P4	37	8.6	132	64	2.34	3	appx3 (64, 3)	36	8.6	121	64	50.00	3	68
16-bit Adders							16-bit Adders							
RCA_N16‡	93	13.4	303	0	0.00	0	RCA_N16‡	93	13.4	303	0	0	0.00	0
ACA_II_N16_Q4 ±	75	7	269	17472	47.79	13	appx8 (17472, 13)	41	7	120	8320	99.64	13	151
ACA_II_N16_Q8	104	10.2	331	4096	5.86	9	appx9 (4096, 9)	94	13.4	254	2038	99.80	9	229
ACA_I_N16_Q4	103	7	321	34944	34.05	13	appx10 (34944, 13)	41	7	120	8320	99.64	13	150
ETAII_N16_Q4 ††	75	7	269	17472	47.79	13	appx8 (17472, 13)	41	7	120	8320	99.64	13	151
ETAII_N16_Q8	104	10.2	331	4096	5.86	9	appx9 (4096, 9)	94	13.4	254	2038	99.80	9	229
GDA_St_N16_M4_P4 ∓	110	10	358	4096	5.86	9	appx9 (4096, 9)	94	13.4	254	2038	99.80	9	229
GDA_St_N16_M4_P8	119	11.1	381	4096	0.18	5	appx11 (4096, 5)	95	13.4	277	496	96.88	5	201
GeAr_N16_R2_P4 ‡‡	81	8.6	284	16640	11.55	11	appx12 (16640, 11)	89	12.7	226	4090	99.90	11	187
GeAr_N16_R4_P4	104	10.2	331	4096	5.86	9	appx9 (4096, 9)	94	13.4	254	2038	99.80	9	229
GeAr_N16_R4_P8	89	11.8	301	4096	0.18	5	appx11 (4096, 5)	95	13.4	277	496	96.88	5	201
GeAr_N16_R6_P4	114	10.2	375	1024	3.08	7	appx13 (1024, 7)	94	13.4	264	1024	99.22	7	220

⋆ As reported by ABC [17] with library *mcnc.genlib*. Area reports normalized to *INVX1*
† $e_{\text{wc-in}}$ and $e_{\text{bf-in}}$ are the error criteria (e_{wc} and e_{bf}) given as input to the tool.

‡, ±, ∓, ‡‡, †† Abbreviations are as given in the online repository : http://ces.itec.kit.edu/1025.php [7]
‡ RCA_N8 and RCA_N16 are 8-bit and 16-bit *ripple carry adders*. These adders are the non-approximated reference designs.
± ACA is *Almost Correct Adder* [9], ∓ GDA is *Gracefully Degrading Adder* [30]
‡‡ GeAr is *Generic Accuracy Configurable Adder* [21], †† ETA is *Error Tolerant Adder* [31]

5. EXPERIMENTAL RESULTS

We have implemented all algorithms in C++ as part of our formal verification package.[3] The program reads Verilog RTL descriptions of the the approximated and non-approximated design using Yosys [28] to create the approximation miter. We use ABC [17] to perform equivalence checking of the miter. The experiments are carried out on an Octa-Core Intel Xeon CPU with 3.40 GHz and 32 GB memory running Linux 4.1.6.

In this section, we propose the results of two experimental evaluations. First, we compare the quality of approximate adders synthesized with our approach to state-of-the-art manually architectured approximated adders. Second, we demonstrate the generality and scalability of the approach by applying it to various designs including standard synthesis benchmark networks such as LGSynth91 [29].

5.1 Approximate synthesis of adders

Approximate synthesis is carried out for adder circuits with a high effort level. The results are given in Table 1. These are compared with architecturally approximated adder designs from the repository [7]. Many of these architectures are specifically hand crafted to improve the delay of the circuit.

The case study is carried out as follows. The adders from [7] are evaluated for worst-case error and bit-flip error, and then synthesis is carried out by specifying these values as limits, hence, the synthesis result obtained from our approach cannot be worse. The error-rate is left unspecified and synthesis is allowed to capitalize on this.

The left side of Table 1 lists the error metrics for architecturally approximated adders, evaluated as given in Section 4.4. The performance metrics such as delay and area are compared with the non-approximated *Ripple Carry Adder* (RCA). The same RCA circuit is given as the input to the approximation synthesis tool along with the e_{wc} and e_{bf} achieved with the architecturally approximated schemes. The synthesized circuits are subsequently evaluated for the error metrics to get the achieved synthesis numbers.

For most of the approximation schemes, our synthesis approach is able to generate circuits with a better area and closer delay compared to the architecturally approximated counterparts, at the cost of error-rate. A large number of schemes such as appx2, appx4, appx5, appx8, and appx10 have significantly improved area with delay numbers match-

[3]The package and the benchmarks are available at the repository https://gitlab.com/arunc/approx_synthesis.git

ing those of architectural schemes.[4] This study demonstrates that our automatic synthesis approach can compete with the quality obtained from handcrafted architecturally designs.

5.1.1 *Image Processing Application*

In order to confirm the quality results of the proposed approach, we show their usage in a real-world image compression application. We have used the OpenCores image compression project [13] to study the impact of approximation adders in signal processing. The experimental setup is as follows. The adders in the color space transformation module of the image compression circuit are replaced with the approximation adders synthesized using our approach and some of the architecturally approximated adders. The input image is the well-known standard test image taken from Wikipedia,[5] trimmed to the specific needs of the image compression circuits. The images generated using these circuits are compared with the non-approximated design using ImageMagick.[6] These images are shown in the Table 2. Only the image obtained with *appx-50k* (an approximate adder synthesized with $e_{\text{wc-in}}$ set to 50000) is heavily distorted. All other generated images may still be considered as of acceptable quality depending on the specific use case. For comparison, we used ACA_II_N16_Q4 and ETAII_N16_Q8 as the architecturally approximated adders.[7] The image quality is comparable to the synthesized approximate adders. Both sets of images do not appear to have a big quality loss despite the high error-rate in approximation synthesis adders. This is due to human perceptual limitations.

A quantitative analysis of the distortions introduced due to approximations can be done using the PSNR (*Peak Signal to Noise Ratio*) plots given in the latter part of Table 2. Using the plots, the difference can be better judged. As can be seen, the synthesized adders show comparable measures to the architectural adders.

In this application case study, the approximation adders are used without considering the features and capabilities of the compression algorithm in depth. A detailed study of approximation adders in the context of image processing is given in [22, 8, 16].

5.1.2 *Note on Error-Rate*

As can be seen from the results in Table 1, the synthesized approximated adders have a higher error-rate. However, this has no effect on the quality in many scenarios, as, e.g., shown in the image compression case study. The error-rate is a metric that relates to the number of errors introduced as a result of approximation. In many signal processing applications involving arithmetic computations (e.g., image compression), designers may choose to focus on other error metrics such as worst-case error [12]. Since the decision is already taken to introduce approximations, the *impact* or the magnitude of errors could be of more significance than the absolute total number of errors itself. Besides, for a general sequential circuit, errors tend to *accumulate* over a period of operation. Though it may be argued that circuits with higher error-rate have higher chance of accumulating errors, in practice, this is strongly dependent on the *composition* of the circuit

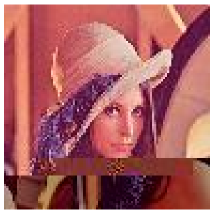
[4]This can be seen by a line-by-line comparison.
[5]https://en.wikipedia.org/wiki/Lenna
[6]http://www.imagemagick.org/
[7]We use the naming convention given in the repository [7].

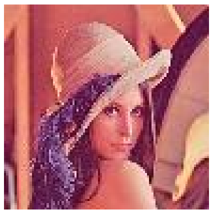
Table 2: Image Processing with Approximation Adders

appx12

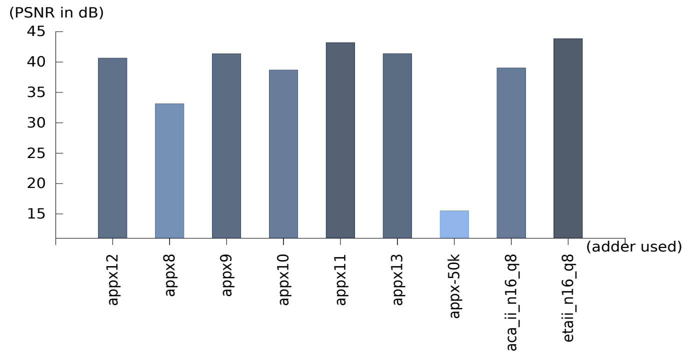
appx8

appx9


appx-50k


ACA_II_N16_Q4

ETAII_N16_Q8



Approximation adder schemes vs PSNR achieved

itself and the input data. Further details on estimating the impact of errors in sequential circuits can be found in [4]. Nevertheless, there is a broad range of applications where error-rate is an important metric in the design of approximate hardware [3, 12].

5.2 Generality and Scalability

We evaluated our method for a wide range of designs and benchmark circuits. The results given in the Table 3 show the generality and applicability of our method. A subset of the LGSynth91 [29] circuits are given in the left side of the table. Each circuit is synthesized in three flavors: (i) specifying values of all the error metrics together, (ii) specifying only the error-rate, and (iii) specifying both worst-case error and bit-flip error, leaving out error-rate. The achieved delay and area in these three schemes are compared with the original non-approximated circuit given as the first entry in each section. In a similar way, several other circuits such as multipliers (Array, Wallace tree, and Dadda tree) and multiply accumulators (MACs) are given next. Besides these standard arithmetic designs, other circuits such as parity generator, priority encoder, and BCD converters are also synthesized and the results are given in Table 3. In almost all the cases, the tool is able to optimize area exploiting the flexibility in the provided error limits. In many cases, delay is also simultaneously optimized along with area. As

Table 3: Approximate-Synthesis Results

LGSynth91 benchmark circuits

Design[†]/Synthesis[††] ($e_{\text{wc-in}}$, $e_{\text{bf-in}}$, $e_{\text{er-in}}$)[††]	Gates⋆	Delay⋆ (*ns*)	Area⋆	e_{wc}	e_{er} (%)	e_{bf}	time (*sec*)
cm163a (I:16,O:5)	34	5.70	78	0	0	0	0
*appx*1 (16, 3, 50)	15	4.10	25	14	43	3	7
*appx*2 (-1, -1, 25)	18	3.00	36	14	21	3	1
*appx*2 (10, 2, -1)	20	4.70	41	8	88	2	4
z4ml (I:7,O:4)	31	12.10	84	0	0	0	0
*appx*1 (8, 1, 75)	20	8.70	52	2	50	1	3
*appx*2 (-1, -1, 25)	31	12.10	84	0	0	0	7
*appx*3 (4, 3, -1)	5	3.80	13.00	4	82	2	3
alu2 (I:10,O:6)	259	32.20	627	0	0	0	0
*appx*1 (30, 3, 50)	236	32.20	570	16	45	2	37
*appx*2 (-1, -1, 25)	231	32.20	566	16	24	1	1
*appx*3 (20, 6, -1)	8	3.30	16	19	81	3	12
frg1 (I:28,O:3)	129	27.10	321	0	0	0	0
*appx*1 (3, 2, 50)	128	27.10	317	1	44	1	10
*appx*2 (-1, -1, 25)	126	27.10	313	2	16	1	1
*appx*2 (2, 1, -1)	128	27.10	317	1	56	1	3
alu4 (I:14,O:8)	519	40.00	1247	0	0	0	0
*appx*1 (128, 4, 50)	489	40.00	1172	64	22	1	139
*appx*2 (-1, -1, 25)	489	40.00	1172	64	22	1	1
*appx*3 (80, 6, -1)	239	34.70	557	79	95	5	55
unreg (I:36,O:16)	83	3.40	227	0	0	0	0
*appx*1 (32000, 8, 50)	80	3.40	214	512	38	1	45
*appx*2 (-1, -1, 25)	83	3.40	227	0	0	0	1
*appx*3 (10000, 12, -1)	46	3.40	90	9088	99	10	17
x2 (I:10,O:7)	30	5.80	74	0	0	0	0
*appx*1 (64, 4, 50)	23	5.70	53	64	37	3	7
*appx*2 (-1, -1, 25)	27	5.70	66	64	12	1	1
*appx*3 (50, 6, -1)	17	5.60	41	40	1	3	9
count (I:35,O:16)	120	14.60	261	0	0	0	0
*appx*1 (32000, 8, 50)	104	14.60	220	7	43	3	140
*appx*2 (-1, -1, 25)	53	3.00	110	65535	24	16	1
*appx*3 (10000, 12, -1)	101	14.40	209	8	97	4	141
term1 (I:34,O:10)	142	11.10	336	0	0	0	0
*appx*1 (500, 5, 50)	113	11.10	249	306	38	5	57
*appx*2 (-1, -1, 25)	84	8.10	179	1012	25	7	1
*appx*3 (300, 8, -1)	73	11.10	150	278	99	8	35

Other Designs

Design[†]/Synthesis[††] ($e_{\text{wc-in}}$, $e_{\text{bf-in}}$, $e_{\text{er-in}}$)[††]	Gates⋆	Delay⋆ (*ns*)	Area⋆	e_{wc}	e_{er} (%)	e_{bf}	time (*sec*)
Multipliers and MAC[‡]							
ArrayMul (I:16,O:16)	420	33.40	1193	0	0	0	0
*appx*1 (32000,4,50)	420	33.40	1188	128	49	1	759
*appx*2 (-1,-1,25)	435	31.60	1234	256	24	8	1
*appx*3 (20000,14,-1)	404	33.20	1086	16448	99	9	107
WallaceMul (I:16,O:16)	398	33.50	1156	0	0	0	0
*appx*1 (32000, 4, 50)	397	33.50	1146	512	49	1	1055
*appx*2 (-1,-1, 25)	391	31.50	1142	512	23	7	3
*appx*3 (20000, 14, -1)	352	33.50	1029	5952	99	10	82
DaddaMul (I:16,O:16)	383	30.20	1082	0	0	0	0
*appx*1 (32000, 4, 50)	382	30.20	1072	1024	47	1	1145
*appx*2 (-1, -1, 25)	368	30.20	1047	832	24	10	7
*appx*3 (20000, 14, -1)	331	30.20	933	2368	99	10	78
Mac8 (I:24,O:16)	434	32.30	1237	0	0	0	0
*appx*1 (32000, 4, 50)	433	32.30	1227	256	49	1	1483
*appx*2 (-1, -1, 25)	421	32.30	1208	768	23	8	79
*appx*3 (20000, 8, -1)	411	32.30	1127	16704	99	8	135
Mac32 (I:48,O:33)	519	48.10	1506	0	0	0	0
*appx*1 (-1, -1, 25)	485	34.20	1371	65536	24	17	1136
Parity[◇] (I:32,O:36)	136	13.00	276	0	0	0	0
*appx*1 (-1, 1, -1)[∗∗]	111	13.00	215	4G	50	1	29
*appx*2 (-1, 2, -1)[∗∗]	86	13.00	154	12G	75	2	24
*appx*3 (-1, 3, -1)[∗∗]	61	13.00	94	30G	88	3	24
Priority[‡‡] (I:32,O:36)	96	26.30	225	0	0	0	0
*appx*1 (1, -1, -1)	78	19.60	176	1	83	1	168
*appx*2 (4, -1, -1)	45	16.90	91	4	96	3	69
*appx*3 (10, -1, -1)	43	12.10	94	8	40	4	12
Bin2BCD[±] (I:8,O:10)	240	30.20	563	0	0	0	0
*appx*1 (-1, -1, 10)	231	28.00	529	576	6	2	1
*appx*2 (-1, -1, 20)	229	28.00	524	576	19	5	1
*appx*3 (-1, -1, 30)	214	27.50	492	110	28	7	1
BCD2Bin[∓] (I:10,O:8)	64	16.10	209	0	0	0	0
*appx*1 (10, -1, -1)	62	16.10	194	8	9	3	33
*appx*2 (25, -1, -1)	62	16.10	189	22	93	4	31
*appx*3 (50, -1, -1)	61	16.10	182	46	97	5	19

† Design name given with primary-inputs (I) and primary-outputs (O) in parenthesis.
†† $e_{\text{wc-in}}$, $e_{\text{bf-in}}$ and $e_{\text{er-in}}$ are the error criteria (e_{wc}, e_{bf}, e_{er}) given as input to the tool. Value -1 indicates that this metric is not enforced. $e_{\text{er-in}}$ is specified as a percentage value. Synthesized output circuits are *appx*1, *appx*2 and *appx*3. Tool runs are with effort level *low*.
⋆ As reported by ABC [17] with library *mcnc.genlib*. Area reports normalized to *INVX1*

‡ Multipliers and multiply accumulate (MAC) designs are generated from : http://www.aoki.ecei.tohoku.ac.jp/arith [1].
◇ Parity Generator, (4 bits parity, 32 bits data). ∗∗ G stands for a multiplier of 10^9. Numerical precision omitted for brevity.
‡‡ 32 to 5 Priority Encoder. ±, ∓ Binary to BCD and BCD to Binary converters; 3 digit BCD and 10 bit binary.

a consequence, the synthesis approximated circuits have a substantially improved *area-delay* product value. In general, as the circuit size (area and gate count) reduces, the power consumed by the circuit also decreases. Hence these approximated circuits also benefit from reduced power consumption.

6. CONCLUSIONS

In this paper, we proposed an automatic synthesis approach for approximate circuits using AIG-based rewriting as underlying technique. Our method can synthesize high quality approximation circuits within user-specified error bounds for worst-case error, bit-flip error, and error-rate. Experimental evaluation on several applications confirm that our methodology has a large potential and the synthesized circuits are even comparable to hand-crafted architecturally approximated circuits in quality. Besides, we presented case studies where the ability of our method to significantly improve circuit performance, capitalizing on less significant error criteria while respecting more stringent ones, is demonstrated.

In future work we want to *increase quality* by finding better strategies for cut replacement and path selection. Furthermore, we want to *increase applicability* by restricting the rewriting to user-defined parts of the circuit. The user can then specify, e.g., to only approximate the data path but not change the control path.

Acknowledgments. This work was supported by the German Research Foundation (DFG) in the project MANIAC (DR 287/29-1), by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative, by the German Federal Ministry of Education and Research (BMBF) in the project EffektiV (01IS13022E), by H2020-ERC-2014-ADG 669354 CyberCare, and by the German Academic Exchange Service (DAAD).

References

[1] Aoki Laboratory - Graduate School of Information Sciences. Tohoku University, 2016. http://www.aoki.ecei.tohoku.ac.jp/arith.

[2] A. Bernasconi and V. Ciriani. 2-SPP approximate synthesis for error tolerant applications. In *EUROMICRO Symposium on Digital System Design*, pages 411–418, Aug 2014.

[3] M. Breuer. Determining error rate in error tolerant VLSI chips. In *Electronic Design, Test and Applications*, pages 321–326, Jan 2004.

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

[5] J. Cong and Y. Ding. Combinational logic synthesis for lut based field programmable gate arrays. *ACM Trans. Des. Autom. Electron. Syst.*, 1(2):145–204, Apr. 1996.

[6] N. Een. Cut sweeping. *Cadence Design Systems, Tech. Rep*, 2007.

[7] GeAr-ApproxAdderLib. Chair for Embedded Systems - Karlsruhe Institute of Technology, 2015. http://ces.itec.kit.edu/1025.php.

[8] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy. Impact: Imprecise adders for low-power approximate computing. In *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, pages 409–414, Aug 2011.

[9] A. Kahng and S. Kang. Accuracy-configurable adder for approximate arithmetic designs. In *Design Automation Conf.*, pages 820–825, June 2012.

[10] P. Kulkarni, P. Gupta, and M. Ercegovac. Trading accuracy for power with an underdesigned multiplier architecture. In *VLSI Design*, pages 346–351, Jan 2011.

[11] N. Li and E. Dubrova. AIG rewriting using 5-input cuts. In *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pages 429–430, Oct 2011.

[12] A. Lingamneni, C. Enz, J. L. Nagel, K. Palem, and C. Piguet. Energy parsimonious circuit design through probabilistic pruning. In *2011 Design, Automation Test in Europe*, pages 1–6, March 2011.

[13] D. Lundgren. OpenCore JPEG Encoder - OpenCores community, 2016. http://opencores.org/project,jpegencode.

[14] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *, IEEE International Conference on Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers*, pages 6–9.

[15] J. Miao, A. Gerstlauer, and M. Orshansky. Approximate logic synthesis under general error magnitude and frequency constraints. In *International Conference on Computer-Aided Design*, pages 779–786. IEEE, 2013.

[16] J. Miao, K. He, A. Gerstlauer, and M. Orshansky. Modeling and synthesis of quality-energy optimal approximate adders. In *2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 728–735, Nov 2012.

[17] A. Mishchenko, M. Case, R. K. Brayton, and S. Jang. Scalable and scalably-verifiable sequential synthesis. In *International Conference on Computer-Aided Design*, pages 234–241, Nov 2008.

[18] A. Mishchenko, S. Chatterjee, and R. K. Brayton. Dag-aware aig rewriting a fresh look at combinational logic synthesis. In *Proceedings of the 43rd Annual Design Automation Conference*, DAC '06, pages 532–535, New York, NY, USA, 2006. ACM.

[19] P. Pan and C.-C. Lin. A new retiming-based technology mapping algorithm for LUT-based FPGAs. In *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, FPGA '98, pages 35–42, New York, NY, USA, 1998. ACM.

[20] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan. ASLAN: Synthesis of approximate sequential circuits. In *Design, Automation and Test in Europe*, pages 1–6, March 2014.

[21] M. Shafique, W. Ahmad, R. Hafiz, and J. Henkel. A low latency generic accuracy configurable adder. In *Design Automation Conf.*, pages 1–6, June 2015.

[22] D. Shin and S. Gupta. Approximate logic synthesis for error tolerant applications. In *Design, Automation and Test in Europe*, pages 957–960, March 2010.

[23] D. Shin and S. K. Gupta. A new circuit simplification method for error tolerant applications. In *Design, Automation and Test in Europe*, pages 1–6, 2011.

[24] M. Soeken, D. Große, A. Chandrasekharan, and R. Drechsler. BDD minimization for approximate computing. In *ASP Design Automation Conf.*, 2016.

[25] http://baldur.iti.kit.edu/sat-race-2015/. SAT-Race 2015, International Conference on Theory and Applications of Satisfiability Testing, 2015.

[26] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan. Salsa: Systematic logic synthesis of approximate circuits. In *Design Automation Conf.*, pages 796–801, June 2012.

[27] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan. MACACO: modeling and analysis of circuits for approximate computing. In *International Conference on Computer-Aided Design*, pages 667–673, 2011.

[28] C. Wolf. Yosys - Yosys Open SYnthesis Suite, 2015. http://www.clifford.at/yosys/about.html.

[29] S. Yang. Logic synthesis and optimization benchmarks user guide version 3.0, 1991.

[30] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu. On reconfiguration-oriented approximate adder design and its application. In *International Conference on Computer-Aided Design*, pages 48–54, Nov 2013.

[31] N. Zhu, W. L. Goh, and K. S. Yeo. An enhanced low-power high-speed adder for error-tolerant application. In *Integrated Circuits, ISIC '09. Proceedings of the 2009 12th International Symposium on*, pages 69–72, Dec 2009.