MUSCAT: MUS-based Circuit Approximation Technique

Linus Witschen, Tobias Wiersema, Matthias Artmann, Marco Platzner
Paderborn University, Germany
Email: {witschen, wiersema, martmann, platzner}@mail.upb.de

***Abstract*—Many applications show an inherent resiliency against inaccuracies and errors in their computations. The design paradigm approximate computing exploits this fact by trading off the application's accuracy against a target metric, e.g., hardware area. This work focuses on approximate computing on the hardware level, where approximate logic synthesis seeks to generate approximate circuits under user-defined quality constraints.**

We propose the novel approximate logic synthesis method MUSCAT to generate approximate circuits which are valid-by-construction. MUSCAT inserts cutpoints into the netlist to employ the commonly-used concept of substituting connections between gates by constant values, which offers potential for subsequent logic minimization. MUSCAT's novelty lies in utilizing formal verification engines to identify minimal unsatisfiable subsets. These subsets determine a maximal number of cutpoints that can be activated together without resulting in a violation against the user-defined quality constraints. As a result, MUSCAT determines an optimal solution w.r.t. the number of activated cutpoints while providing a guarantee on the quality constraints.

We present the method and experimentally compare MUSCAT's open-source implementation to AIG rewriting and components from the EvoApproxLib. We show that our method improves upon these state-of-the-art methods by achieving up to 80 % higher savings in circuit area at typically much lower computation times.

***Index Terms*—Approximate Computing; Formal Verification; Approximate Logic Synthesis; Approximate Circuit Synthesis; Minimal Unsatisfiable Subsets**

I. Introduction

Many applications exhibit error resiliency, e.g., image and video processing, signal processing, or machine learning [1]. The design paradigm approximate computing exploits this fact to trade off an application's quality against improvements in a target metric, e.g., hardware area or energy consumption. To realize this trade-off, approximate computing techniques modify code or logic to introduce errors in the system judiciously. While the errors are limited to user-defined quality constraints, the system's performance can be improved significantly.

This work focuses on approximate logic synthesis (ALS) with the primary goal to generate approximate logic circuits (AxCs) that are optimized for a target metric while satisfying user-defined quality constraints expressed in terms of error metrics, e.g., the worst-case error. In their survey, Scarabottolo et al. [2] describe the three components of a general ALS procedure as error modeling, the actual ALS method, and quality-of-result evaluation (*quality verification*). The error model is created for a specific input design and used by the actual ALS method to quickly estimate the impact of an approximation on the overall circuit error. Quality verification is needed to verify that an AxC is valid, i.e., adheres to the quality constraints. While error models can benefit the approximate outcome [3], an open issue is that these models are often inexact and too conservative [2], leading to unused approximation potential. Furthermore, a dedicated quality verification applied from time to time may force the ALS to back-track and revoke approximations to establish an AxC that is valid.

In this paper, we present the novel ALS technique MUSCAT, a **MUS**-based **C**ircuit **A**pproximation **T**echnique. MUSCAT creates AxCs that are *valid-by-construction* regarding their quality constraints and, thus, neither requires error models nor dedicated quality verification. Our ALS technique bases on the commonly-used concept of substituting connections between gates by constant values [3], [4]. To achieve this, MUSCAT firstly augments an input netlist by so-called *cutpoints*, which, if activated, perform the substitution. Then, MUSCAT employs the novel approach of utilizing formal verification engines to determine *minimal unsatisfiable subsets* (MUSes). The MUSes identify the maximal number of cutpoints that can be activated together safely, i.e., the constructed AxC is guaranteed to adhere to the quality constraint. Activating any further cutpoints would render the AxC invalid. Hence, a MUS specifies an optimal solution w.r.t. the number of activated cutpoints, and our practical experiments show that after applying standard logic synthesis tools to subsequently minimize the circuit we indeed achieve significant improvements over state-of-the-art techniques. Since the use of formal methods is often questioned, it is important to note that they have matured significantly in the recent past and can verify designs with millions of gates and hundreds of inputs in a reasonable time [5], [6].

In summary, our contributions are:

- We propose MUSCAT, a novel MUS-based ALS technique that exploits modern formal verification engines to generate AxCs with quality constraints valid-by-construction. Our technique does not require an error model and thus overcomes limitations of previous work, in particular it allows for supporting any non-statistical error metric.
- We present experiments showing that, compared to state-of-the-art ALS, our method achieves up to 80% higher area savings.
- MUSCAT is open-source and publicly available.

We structure the remainder of this paper as follows: Section II discusses related work. Section III presents our novel ALS technique, and Section IV discusses MUSCAT's experimental results. Finally, Section V concludes the paper.

978-3-9819263-6-1/DATE22/©2022 EDAA

II. Discussion of the State-of-the-Art

Scarabottolo et al. [2] distinguish between three categories of ALS methods: i) netlist transformations that approximate the input design's gate-level netlist, ii) Boolean rewriting that operates on a design's abstract representation, e.g., truth tables, and iii) approximate high-level synthesis that starts with behavioral level descriptions. Our approach relates to categories i) and ii) and, thus, we review corresponding related work in this section.

A main approach for netlist transformation is gate-level pruning (GLP) presented by Schlachter et al. [3], [7]. A wire is disconnected from the node driving it and, instead, a constant is inserted as driver. As a result, a synthesis tool can optimize the logic by 1) removing (or pruning) dangling nodes and 2) simplifying the subsequent logic through constant propagation. To determine which node to prune, the authors rank the nodes by the product of their switching activity and their significance, a measure for the node's impact on the circuit's error. An iterative algorithm simulates the hardware design to compute each node's significance-activity product and to evaluate whether the design meets the quality constraints. The algorithm then prunes the node with the lowest significance-activity product from the design in the next iteration. Scarabottolo et al. [8], [9] show that a node's significance is often too conservative, leading to sub-optimal AxCs.

Thus, Scarabottolo et al. [8] extended GLP to circuit carving. Circuit carving seeks to carve out the largest subcircuit in the design so that the resulting AxC still meets the quality constraints. The authors propose to use a node's significance as an error estimate and explore a binary search tree to determine whether a node is included in the subcircuit. However, as an AxC's actual error cannot be determined via significances, a subsequent quality check is performed. An exploration of the complete binary search tree is considered intractable; hence, the authors defined pruning criteria for the tree. One criterion considers the estimated significance (or estimated error) of a node. To estimate the nodes' significance, the authors suggest simulating the complete circuit exhaustively, which is accurate but only applicable to small circuits, or employing an error estimation model, which has shown to be often too conservative. In any case, the error estimation's accuracy dictates the approximate outcome.

In an attempt to increase the error estimation accuracy in GLP, Partition & Propagate (P&P) [9] was proposed. P&P partitions the design into cuts which are simulated exhaustively to determine each node's significance. Due to the exhaustive simulation, the accuracy of the nodes' significance becomes more accurate. The authors argue that the number of inputs to the cuts is small, and thus, simulation can be performed efficiently. Nevertheless, the approach can only specify an estimate of an error bound, which, as their experimental results show, may still be too conservative by several orders of magnitudes. Consequently, a subsequent quality verification is required.

Another netlist transformation approach is circuit design by evolutionary algorithms that iterate over thousands of circuit generations and modify netlists by mutation operators. This approach often leads to unusual, yet efficient AxCs as, for example, shown by the adders and multipliers provided in the EvoApproxLib [10].

Boolean rewriting techniques approximate Boolean representations of a circuit, such as truth tables or and-inverter graphs (AIGs). BLASYS [11] relies on Boolean matrix factorization, and factorizes a circuit's truth table into two smaller truth tables, which results in a smaller AxC after synthesis. Considering complete truth tables, however, limits BLASYS to small circuits. Thus, for larger circuits, BLASYS firstly factorizes subcircuits with every factorization degree. A search then integrates the approximated subcircuits into the overall design until the user-defined error threshold is reached. ALSRAC [12] utilizes statistical error metrics, such as the error rate, and relies on approximate care sets, determined via logic simulation and expressed for internal nodes. Using the approximate care patterns, ALSRAC generates approximate resubstitutions, which are applied iteratively until the error threshold is reached. Chandrasekharan et al. [4] approximate an AIG representation of a design. First, the nodes on the critical paths are identified and sorted by their cut size. Then, the nodes are replaced iteratively by a constant, starting with the smallest cut. SALSA [13] reverses the approximation process by setting up a miter structure that employs a quality constraint function with the exact and the AxC at the inputs. Assuming that the quality constraints must be satisfied, some of the outputs of the AxC become don't cares and by subsequent standard don't care optimization the AxC can be simplified. In this way, SALSA generates AxCs that the authors denote as correct-by-construction, which essentially means the same as valid-by-construction, making a subsequent quality verification step obsolete.

Our MUS-based approach applies approximations via netlist transformations similar to GLP [3] and also resembles the concept of cuts used in AIG rewriting [4]. However, our approach is fundamentally different from related work in netlist transformation and most approaches in Boolean rewriting since we generate AxCs that are valid-by-construction without need for an error model or dedicated quality verification steps. The independence from an error estimation or propagation model enables our approach to support any non-statistical error metric, while GLP and its successors only provide an error model for the worst-case error. Furthermore, AIG rewriting considers only nodes on the critical paths – usually carry-chains in arithmetic components –, making the approach often too aggressive (see [14] and Section IV). Comparing MUSCAT to SALSA, that also spares error modeling and dedicated quality verification steps, we identify two major differences in the employed ALS technique and in the abstraction level. MUSCAT operates on the netlist level, and inserts cutpoints and computes minimal unsatisfiable subsets, while SALSA is a Boolean rewriting approach and relies on standard don't care optimization.

III. Methodology

We represent the gate-level netlist of a combinational circuit as a directed acyclic graph (DAG) $G(N, E)$, where $n \in N$ represent the nodes (gates) in G, and $(a, b) \in E$ represent the edges (connections between gates). For an edge (a, b), node a's

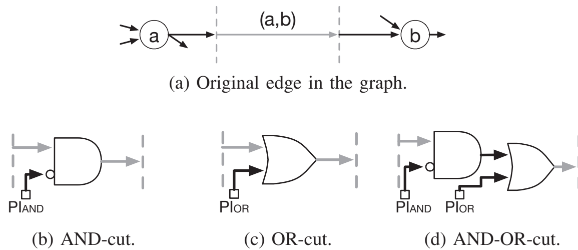
output drives node b's input. We augment a given DAG G with so-called *cutpoints*. A cutpoint is inserted at an edge (a, b) and consists of one or two gates by which we can either *activate* or *deactivate* the cutpoint. An *activated* cutpoint ruptures the connection and drives b's input by either constant 0 or 1; a *deactivated* cutpoint leaves the connection unchanged, i.e., the original signal from node a can propagate to node b.

For a given G, we denote the set of all cutpoints as C. A *cutpoint configuration* (short: *configuration*) defines for each cutpoint $c \in C$ whether it is activated or deactivated. Activated cutpoints offer optimization potential and subsequent logic synthesis tools will remove dangling nodes and simplify the logic through constant propagation.

In the following, we first discuss different cutpoint designs and the approximation miter for determining and verifying configurations. Then, we elaborate on *minimal unsatisfiable subsets*, which we use as a specification for configurations. Finally, we present our ALS algorithm and discuss the cutpoint insertion.

A. Cutpoints

Fig. 1a shows a segment from a graph $G(N, E)$ with the nodes $a, b \in N$ and the edge $(a, b) \in E$. The gray, vertically-dashed lines indicate the insertion of a cutpoint $c \in C$ for the edge (a, b). Figs. 1b to 1d present different cutpoint designs.



(a) Original edge in the graph.

(b) AND-cut. (c) OR-cut. (d) AND-OR-cut.

Fig. 1. Concept of replacing an edge with a cutpoint in a netlist. Dashed gray lines indicate the cut, solid gray lines indicate the original connection, and boxes represent newly added primary inputs.

The AND-cut in the edge (a, b) as shown in Fig. 1b comprises an AND gate with one input connected to the output of a and the second one is exposed as inverted new primary input (PI) PI_{AND}. Depending on the value for this new primary input, the edge remains either unchanged or is cut with $PI_{AND} = 1$ and the logic value 0 drives the input of node b. The OR-cut shown in Fig. 1c inverses the AND-cut's behavior, i.e., $PI_{OR} = 1$ activates the OR-cut and propagates a constant 1 to the input of node b. The AND-OR-cut displayed in Fig. 1d allows for propagating either the constants 0 or 1 when activated but exposes two new primary inputs PI_{AND} and PI_{OR} to realize the cutpoint. While the AND-cut and the OR-cut are alternatives that in general will create different optimization potential for subsequent logic synthesis, the AND-OR-cut subsumes the functionality of both single gate cuts at the expense of doubling the number of newly exposed PIs.

B. Approximation Miter

Our ALS technique determines a suitable configuration for a given set of cutpoints C and simultaneously verifies the validity of the resulting AxC by formulating a satisfiability (SAT) problem in the SMT domain. To that end, we construct an approximation miter as shown in the example of Fig. 2 for a circuit with 6 nodes, 5 PIs, 3 primary outputs (POs), and a set C of 16 possible cuts. The approximation miter comprises of the exact circuit, the AxC including the cutpoints, and verification logic. The verification logic determines the circuit's error ϵ by comparing the POs of the AxC with the POs of the exact circuit and checking whether ϵ exceeds a given threshold T that corresponds to the specified quality constraint. If the chosen configuration of cutpoints leads to an error exceeding T, the miter becomes satisfiable and the output of the miter will be raised, indicating a quality constraint violation. Consequently, if we prove the unsatisfiability (UNSAT) of the approximation miter we have a guarantee that the chosen configuration actually leads to an AxC adhering to the quality constraint.

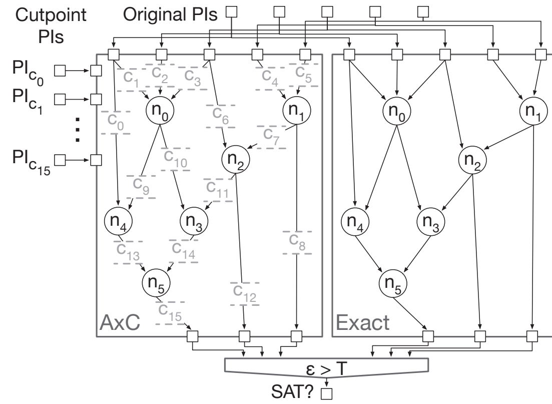


Fig. 2. Approximation miter with the cutpoints $C = \{c_0, ..., c_{15}\}$ and the corresponding control inputs PI_c.

C. Minimal Unsatisfiable Subsets

Under the assumption that an activated cutpoint leads to removal of logic and thus to savings in the target metric, we can state that by activating more cutpoints the savings will generally increase. Hence, we are interested in finding configurations with as many cutpoints activated as possible while still respecting the quality constraint. That can be cast as *minimal unsatisfiable subset* (MUS) problem: Given an UNSAT problem with a set of constraints Q, $M \subseteq Q$ is a MUS of Q iff M is unsatisfiable and for all $m \in M$ the set $M \setminus \{m\}$ is satisfiable [15]. Note that there can be several MUSes with different cardinality.

Transferred to our ALS setting, we maintain the original primary inputs of the circuit as free variables that a SMT solver can assign, and use the additional primary inputs driving the cutpoints as constraints. When setting as constraints $PI_c = 0, \forall c \in C$, the AxC resembles the exact circuit and the corresponding approximation miter will be unsatisfiable since

the error is zero. Now, solving the MUS problem will return a minimal set of constraints M required to keep unsatisfiability. Each of these constraints $m \in M$ corresponds to a cutpoint that must be deactivated. Vice versa, the set of remaining constraints, which is maximally large, is not required to maintain UNSAT and thus the corresponding cutpoints will be activated to reduce the logic.

In other words, we initially constrain each cutpoint's PI to be deactivated to resemble the exact circuit and construct an UNSAT problem. Then, solving the MUS problem returns a configuration that is guaranteed to be UNSAT and holds a minimal number of cutpoints that have to be deactivated to maintain unsatisfiability. Cutpoints that are not part of the returned configuration can be activated safely.

D. Approximate Logic Synthesis Flow

Algorithm 1 Pseudo code of our ALS technique

```
Input: Exact circuit O, error metric EM, error threshold T
Output: Best AxC
1: AxC, C ← addCutpoints( O )
2: AM ← constructApproximationMiter(O, AxC, EM, T)
3: Q ← createUNSATConstraints(C, Q )
4: MUSes ← determineMUSes( AM, Q )
5: AxCs ← generateAxCs( AxC, MUSes )
6: return bestAxC( AxCs )
```

Algorithm 1 shows the flow for our proposed ALS technique, which takes as inputs the exact circuit O, the used error metric EM, and the error threshold T. First, the algorithm generates the set of cutpoints C and the AxC by augmenting the exact design O with the cutpoints C (line 1). Next, the approximation miter AM as shown in Fig. 2 is set up (line 2). At this point, the additional primary inputs determining the configuration are actually free variables. In the following step (line 3) these variables are constrained to $PI_c = 0, \forall c \in C$, which we denote as UNSAT constraints since they guarantee unsatisfiability. Then, based on the miter AM and the UNSAT constraints Q, the MUSes are computed (line 4). From the constraints in these MUSes we can directly derive deactivated and, subsequently, activated cutpoints to simplify the corresponding AxCs through logic synthesis (line 5). Finally, Algorithm 1 returns the AxC with the highest savings in the target metric.

E. Discussion on the Insertion of Cutpoints

There are several related decisions with respect to cutpoint insertion: How many cutpoints should be inserted, and where in the circuit? Which cutpoint design should be used, AND-cut, OR-cut, or AND-OR-cut? Generally, inserting cutpoints into every edge of the circuit, as done in the example of Fig. 2, and using the AND-OR-cut maximizes the potential for savings in the target metric. However, the increased number of constraints might increase the runtime for solving the MUS problem considerably. Even though formal techniques can verify designs with millions of gates and hundreds of inputs in a reasonable time [5], [6], selecting a sufficiently small number of well-positioned cutpoints is desirable to keep the solver runtimes reasonable. We keep the number of cutpoints sufficiently small while enabling graceful approximations by evaluating a cutpoint's redundancy and immediate savings.

Initially, the input edges of all nodes qualify for cutpoints. In a first step, we evaluate a cutpoint's redundancy based on its appearance in another cutpoint's maximum fanout-free cone (MFFC). Since cutpoints enclosed in the MFFC become obsolete upon activation of the enclosing cutpoint, the enclosed cutpoints pose redundancy and are thus omitted.

Furthermore, as all nodes within the MFFC become dangling, and thus obsolete, upon a cutpoint's activation, the MFFC's size is an indicator for the immediate savings. Thus, we suggest to utilize the MFFC's size as ranking criterion for the cutpoints, if their number should be reduced further. While other heuristics based on, e.g., the size of the fanin or fanout cone, also take into account the logic cone affected by constant propagation, the actual effects on the savings are unknown in advance. Furthermore, such heuristics may have a structural bias. For example, cutpoints closer to the circuit's PIs will likely have a larger fanout cone than the cutpoints closer to the circuit's POs, which introduces a bias towards the cutpoints closer to the PIs. The MFFC, however, is more robust against structural bias, and our experimental results show that the MFFC is indeed a reasonable heuristic (see Section IV-B).

IV. Experimental Evaluation

A. Implementation and Experimental Setup

We have implemented a fully automated tool flow as outlined in Algorithm 1 and made it available as the open-source project MUSCAT[1]. We employ Yosys [16] to read the input design in Verilog and convert it to a technology-independent netlist. Custom Yosys passes add the cutpoints and construct the approximation miter automatically. Then, we translate the approximation miter into SMT-LIBv2 standard format to ease the addition of constraints via named assertions[2] and to provide a unified interface to different SMT solvers. We have experimented with two such solvers, the SMT solver z3 [17] (labeled `z3` in the result plots) that returns the first MUS found and the MUSTOOL [15] that enumerates a set of MUSes within a given time budget (labeled `must_time-budget` in the plots).

From all available cutpoints, we vary the number of inserted cutpoints from 1% to 100% (suffix *_0.01* or *_1.0* in the plots), using the MFFC's size as heuristic for the cutpoint insertion. The target metric is the circuit area, which we determine by technology mapping to the Nangate 45nm Open Cell library with ABC [18]. The experiments have been performed on a compute cluster with nodes equipped with Intel® Xeon E5-2670@2.6GHz and 4GiB main memory.

B. Results and Discussion

We experimentally compare our novel approach MUSCAT against two publicly available state-of-the-art ALS methods: AIG rewriting [4] and, where possible, the designs in the EvoApproxLib [10].

[1] https://git.uni-paderborn.de/muscat/muscat

[2] We can map named assertions directly to the corresponding cutpoints.

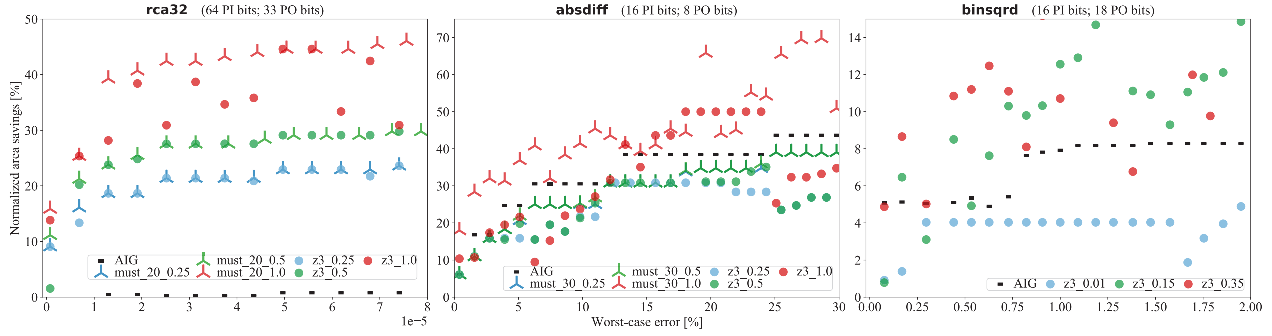


Fig. 3. Comparison of our novel methods (`z3` and `must`) with different cutpoint percentages to `AIG` for three benchmark circuits and varying error bounds.

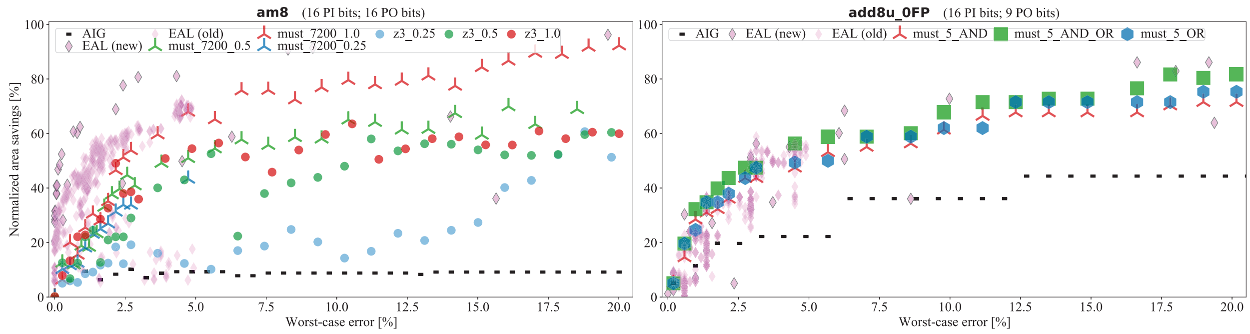


Fig. 4. Comparison of our methods (`z3` and `must`) with different cutpoint percentages to `AIG` and components of the EvoApproxLib (two versions) for two benchmarks and varying error bounds. For the benchmark `add8u_0FP` we use 100% of the cutpoints and vary the cutpoint designs.

Fig. 3 compares MUSCAT using AND-cuts with AIG rewriting for the three benchmark circuits `rca32` (32-bit ripple-carry adder), `absdiff` (absolute difference), `binsqrd` (binomial squared). The plots show the area savings normalized to the exact design's area for different worst-case errors normalized to the circuits' maximum output value. For the adder `rca32`, `AIG` saturates at $\approx 1\%$ savings, while our approach achieves significantly higher savings of up to $\approx 45\%$. Even the setups using `z3` that returns only a single MUS achieve high savings consistently. If cutpoints are inserted in 100% of the edges, `z3_1.0` shows large variations in the savings which are due to the fact that only one MUS is selected. Exploring the MUS search space more thoroughly by producing more MUSes with the setup `must` and a time budget of 20s, MUSCAT minimizes the variations in savings and produces very compact AxCs. Exploring the MUS search space and increasing the number of cutpoints is always beneficial for reducing circuit area.

Comparing the ALS runtimes for this benchmark, `AIG` averages to 343s, `z3` averages to $4, 4.5$ and 7s depending on the percentage of cutpoints, and `must` averages to $18, 46$, and 30s. Clearly, our novel ALS approach is an order of magnitude faster than `AIG` for this benchmark. We can further compare the ALS runtimes for `rca32` with published data from other related work. For example, P&P [9] report a runtime of 18s but only for determining the nodes' significances. The rather costly search and simulations for quality verification are not included in this figure. Both SALSA [13] and our approach create AxCs valid-by-construction. Unfortunately, SALSA is not open source and the related papers do not provide sufficient details for a re-implementation. However, for the same benchmark SALSA reported area savings of $\approx 15\%$ for an error of $1\% * 10^{-5}$ and a runtime of around four minutes. Hence, at this data point we outperform SALSA by achieving savings of $\approx 25\%$ within a few seconds.

For the benchmark `absdiff`, `must` outperforms `AIG`, while in the majority of the cases `AIG` achieves better results than `z3`. `must_30_1.0` achieves the highest area savings, while `must_30_0.25` and `must_30_0.5` appear to saturate, presumably due to a small number of cutpoints. The most challenging circuit is `binsqrd` that contains three multipliers, which are known to be hard to verify [6]. To keep runtimes low, we use only `z3` and consider a smaller percentage of inserted cutpoints. In most cases, MUSCAT achieves better results than `AIG`. However, it can also be seen that limiting the number of cutpoints too aggressively or extracting only one sample from the MUS search space may lead to sub-optimal results, e.g., `z3_0.01` saturates at low savings or the achieved savings of a setup deviate largely, respectively. The runtimes average to $1912, 50842$, and 188105s for different cutpoint percentages. In this case `AIG` shows a shorter average runtime of 1327s.

Fig. 4 shows the results for approximating an 8-bit array multiplier `am8` and an 8-bit adder from the EAL `add8u_0FP` for which components in the EvoApproxLib (`EAL`) are available. `AIG` saturates at low area savings for both benchmarks,

while our approach achieves up to $\approx 80\%$ higher savings. am8 highlights the benefit of must that takes multiple samples from the MUS search space since it achieves higher savings than z3. EAL achieves better area savings, only for larger error bounds our method remains competitive. However, EAL employs an evolutionary design process over thousands of generations that typically requires very long runtimes, but can create efficient designs. For the benchmark add8u_0FP, our area savings are comparable with EAL. In this case we have experimented with different cutpoint designs and show that the AND-OR-cuts with their flexibility achieve the highest area savings.

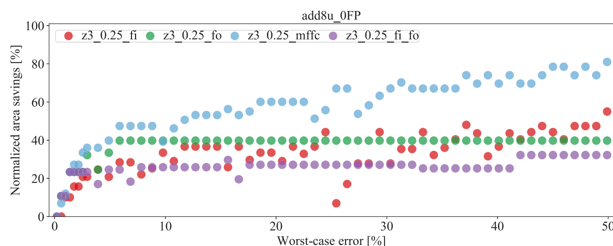


Fig. 5. Comparison of different heuristics for the cutpoint insertion.

Finally, we investigate the impact of the cutpoint insertion on the approximate outcome for the benchmark add8u_0FP in Fig. 5. We have employed z3 and ranked all available cutpoints with different heuristics: the cutpoint's size of the MFFC (mffc), the fanout cone (fo), the fanin cone (fi), as well as the aggregate of the fanin and fanout cones (fi_fo). From the ranked cutpoints, we then only inserted the subset of the best $25\,\%$ cutpoints. The figure shows significant differences in the achieved savings for the heuristics, and justifies ranking cutpoints by their MFFC's size, if only a subset of all available cutpoints should be inserted. The heuristic mffc performs best in the experiment by consistently achieving high savings with small deviations for small as well as large WC errors, i.e., the heuristic enables graceful approximations. The heuristic fo achieves high savings for small WC errors; however, appears to saturate at savings around $40\,\%$ for larger errors. While showing the largest deviations, the heuristic fi improves towards larger error bounds on average. Overall, the heuristic fi_fo performs worst in the experiment.

V. Conclusion

In this paper, we have presented the novel approximate logic synthesis method MUSCAT to generate valid-by-construction approximate circuits (AxCs). Our approach augments an input design by cutpoints that, when activated, allow for logic optimization by removing dangling nodes and by constant propagation. Formulating the AxC's construction as an UNSAT problem, we exploit modern formal verification engines to determine *minimal unsatisfiable subsets* (MUSes) that translate to sets of cutpoints that should be activated. We have experimentally compared MUSCAT to AIG rewriting and components of the EvoApproxLib. MUSCAT achieves comparable or up to 80% better area savings than AIG rewriting at runtimes that are mostly an order of magnitude lower. Furthermore, MUSCAT is competitive with the EvoApproxLib, whose components are designed by a computationally very expensive evolutionary algorithm. We have also compared MUSCAT to SALSA on a design point taken from literature. This comparison indicates that MUSCAT is able to achieve higher area savings at substantially lower runtimes. Our future work includes evaluating more sophisticated heuristics for cutpoint insertion to minimize deviations and maximize savings, assessing the effects of applying MUSCAT to already approximated input designs, and elevating MUSCAT to the register-transfer level.

Acknowledgment

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901). The authors gratefully acknowledge the funding of this project by computing time provided by the Paderborn Center for Parallel Computing.

References

[1] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, 2016.

[2] I. Scarabottolo, G. Ansaloni, G. A. Constantinides, L. Pozzi, and S. Reda, "Approximate logic synthesis: A survey," *Proceedings of the IEEE*, vol. 108, no. 12, 2020.

[3] J. Schlachter, V. Camus, C. Enz, and K. V. Palem, "Automatic generation of inexact digital circuits by gate-level pruning," in *Int. Symp. on Circuits and Systems (ISCAS)*, 2015.

[4] A. Chandrasekharan, M. Soeken, D. Große, and R. Drechsler, "Approximation-aware rewriting of AIGs for error tolerant applications," in *Int. Conf. on Comput.-Aided Des. (ICCAD)*, 2016.

[5] Z. Vasicek, "Relaxed equivalence checking: a new challenge in logic synthesis," in *Int. Symp. on Des. and Diagnostics of Elect. Circuits & Systems (DDECS)*, 2017.

[6] Z. Vasícek, "Formal methods for exact analysis of approximate circuits," *IEEE Access*, vol. 7, 2019.

[7] J. Schlachter, V. Camus, K. V. Palem, and C. Enz, "Design and applications of approximate circuits by gate-level pruning," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 25, 2017.

[8] I. Scarabottolo, G. Ansaloni, and L. Pozzi, "Circuit carving - A methodology for the design of approximate hardware.," in *Des., Automat. & Test in Europe (DATE)*, 2018.

[9] I. Scarabottolo, G. Ansaloni, G. A. Constantinides, and L. Pozzi, "Partition and propagate: an error derivation algorithm for the design of approximate circuits," in *Des. Automat. Conf. (DAC)*, 2019.

[10] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina, "EvoApproxSb: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods," in *Des., Automat. & Test in Europe (DATE)*, 2017.

[11] S. Hashemi, H. Tann, and S. Reda, "BLASYS: Approximate logic synthesis using boolean matrix factorization," in *Des. Automat. Conf. (DAC)*, 2018.

[12] C. Meng, W. Qian, and A. Mishchenko, "ALSRAC: Approximate logic synthesis by resubstitution with approximate care set," in *Des. Automat. Conf. (DAC)*, 2020.

[13] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "SALSA: Systematic logic synthesis of approximate circuits," in *Des. Automat. Conf. (DAC)*, 2012.

[14] L. Witschen, M. Awais, H. G. Mohammadi, T. Wiersema, and M. Platzner, "CIRCA: Towards a modular and extensible framework for approximate circuit generation," *Microelectron. Rel.*, vol. 99, 2019.

[15] J. Bendík and I. Cerná, "MUST: minimal unsatisfiable subsets enumeration tool," in *Int. Conf. on Tools and Algo. for the Constr. and Anal. of Systems (TACAS)*, 2020.

[16] C. Wolf, "Yosys open synthesis suite." http://www.clifford.at/yosys/.

[17] L. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Int. Conf. on Tools and Algo. for the Constr. and Anal. of Systems (TACAS)*, 2008.

[18] Berkeley Logic Synthesis and Verification Group, "ABC: A system for sequential synthesis and verification." http://www.eecs.berkeley.edu/ alanmi/abc/.