Microelectronics Reliability 99 (2019) xxx–xxx

Contents lists available at ScienceDirect

Microelectronics Reliability

journal homepage: www.elsevier.com/locate/microrel

CIRCA: Towards a modular and extensible framework for approximate circuit generation

Linus Witschen, Muhammad Awais, Hassan Ghasemzadeh Mohammadi, Tobias Wiersema, Marco Platzner

Paderborn University, Germany

ARTICLE INFO

Keywords:
Approximate Computing
Framework
Pareto Front
Accuracy
Stochastic search
Circuit synthesis

ABSTRACT

Existing approaches and tools for the generation of approximate circuits often lack generality and are restricted to certain circuit types, approximation techniques, and quality assurance methods. Moreover, only few tools are publicly available. This hinders the development and evaluation of new techniques for approximating circuits and their comparison to previous approaches.

In this paper, we first analyze and classify related approaches and then present CIRCA, our flexible framework for search-based approximate circuit generation. CIRCA is developed with a focus on modularity and extensibility to foster comparability of different techniques. We present the architecture of CIRCA with its distinct separation into stages and processing blocks. This separation makes CIRCA flexible, allowing developers to extend it by new methods and enabling users to exchange the employed functionality quickly. To highlight CIRCA's benefits and its flexibility, we conduct a case study which focuses on the impact of the employed search method and approximation technique. We use a set of circuits from our approximate computing benchmark suite PaderBench to evaluate and compare the quality of the approximated circuits generated with the different methods.

1. Introduction

Approximate computing subsumes a variety of approaches that trade-off computational accuracy for improvements in hardware area, delay, and/or energy consumption. This trade-off becomes possible since for many applications approximate results are indeed acceptable or often hard to distinguish from accurate results. Examples for such applications can be found in the processing of audio, image, and video data as well as in information retrieval and machine learning. While computer scientists and engineers have been dealing with limited accuracy and suboptimal quality of results for a long time, e.g., due to the limited precision of data types or in developing heuristics, recent research has been focusing on more aggressive approximation techniques at all levels of the computing stack, from programming languages down to logic and circuits [1].

Our interest is in methods for the automated approximation of combinational and sequential circuits. The approximation process states an optimization problem with various objectives such as quality, silicon area, and energy consumption. To solve this optimization problem, related approaches presented in literature employ iterative design processes, typically search-based [2-5]. A search space is spanned by iteratively creating different versions of approximate circuits, selecting one of these versions, for example by heuristics, and checking whether the version still satisfies user-provided error bounds. The difficulty in studying and comparing related approaches is that they are designed, or at least described, as monolithic blocks with interwoven phases for search, approximation, and quality checking. Furthermore, only few frameworks are openly available for experimentation [3,4], which hinders the development of new techniques for approximating circuits.

In this paper, we present CIRCA, our framework for approximate circuit generation. CIRCA is developed to be modular, extensible and publicly available[1].

From the analysis and evaluation of existing approximation frameworks (see Section 2), we have found that a comprehensive framework needs to be *general*, *modular*, *compatible*, *extensible*, and *available* in order to be used for the comparison and evaluation of different techniques. Our framework CIRCA, presented in this paper, takes these criteria into account. We argue that such a framework is of great value for the approximate computing community, since it facilitates the rapid implementation and evaluation of new ideas and also enables fair experimental comparisons of alternative techniques. To show the benefits

E-mail addresses: witschen@mail.upb.de (L. Witschen), mawais@mail.upb.de (M. Awais), hgm@mail.upb.de (H. Ghasemzadeh Mohammadi), tobias.wiersema@mail.upb.de (T. Wiersema), platzner@mail.upb.de (M. Platzner).

[1] http://go.upb.de/circa.

https://doi.org/10.1016/j.microrel.2019.04.003
Received 28 September 2018; Received in revised form 22 January 2019; Accepted 2 April 2019

of CIRCA, we have focused on the most important part of the approximate circuit generation, the search-based multi-objective optimization problem. We evaluate the impact of search methods on the approximated outcome by comparing three different search methods, conducting an exemplary case-study in this paper.

Furthermore, we have compiled a benchmark suite, *PaderBench*[2], comprising of 18 circuits from different domains to further encourage and enhance comparability. In this paper, we evaluate our CIRCA framework by performing experiments with seven sequential circuits from our benchmark suite. CIRCA is able to achieve area savings of up to 55 % while guaranteeing that the approximated circuits meet the quality constraints.

In summary, we make the following contributions:

- We develop a general, modular, and extensible approximation framework, CIRCA, which can be easily employed by the research community for investigating, implementing, and verifying emerging approximated circuit generation methods.
- We introduce a benchmark suite for approximate computing called PaderBench which includes 18 circuits from various domains of application such as computer arithmetic, signal processing, and machine learning.
- We demonstrate how extensible CIRCA is by presenting three different search-based optimization methods implemented in the framework. We furthermore conduct a case study on the impact of these methods on the approximated outcome.

The remainder of this paper is structured as follows: in Section 2.1, we discuss related work. In Section 2.2, we provide a classification of current approximate circuit synthesis frameworks. The requirements for a flexible approximation framework, which we derive from the classification, are presented in Section 3. The design of CIRCA is presented in Section 4 and Section 5 discusses the details of implemented methods. Finally, we present our experimental setup, our benchmark suite PaderBench, and the results of our case study, which focuses on search methods, in Section 6. Section 7 concludes the paper and describes future work.

2. Discussion of related work

Approximate computing can be applied to all levels of the computing stack. CIRCA operates on approximations applied during the synthesis process of circuits which can be sub-divided to three levels of abstraction: *behavioral* descriptions, *circuit-level* descriptions, and *logic-level* descriptions. In this paper, we focus on the *circuit* and *logic* levels.

In this section, we first provide a brief overview over related approaches for the automated generation of approximate circuits. We focus on the approximation techniques and how they are iteratively applied to generate one or more approximated circuit variants. Then, we classify the presented frameworks according to their characteristics.

2.1. Overview

Venkataramani et al. proposed SASIMI [2] which uses a substitute-and-simplify approximation technique. Near-identical signal pairs are identified, i.e., two signals which show a similar behavior, and one is substituted with the other. Subsequently, the required logic can be simplified. SASIMI evaluates and ranks signal pairs with a heuristic function including area and delay parameters. With a gradient ascent technique, more accurately hill climbing, the highest ranked pair is selected for substitution. The process is iterated until the user-defined quality constraints are violated. For the resulting circuit, power and area are reported. The quality of the approximate circuits is determined by testing. Additionally, the authors suggest the concept of *quality configurable circuits*, which are circuits that can operate in either an accurate or approximate version.

Venkataramani et al. also presented SALSA [6] which forms a so-called *quality constraint circuit* by providing the original and the approximated circuit, which initially is identical to the original circuit, with the same input and feeding the outputs of the circuits into a quality function that checks whether the given error bound holds. Forcing the error bound to hold, SALSA works backwards and applies standard don't care logic optimization techniques to reduce the area of the approximated circuit. Thus, the technique creates approximated combinational circuits that adhere to the error bound by construction.

Similar to SALSA, the approach of Chandrasekharan et al. [3] employs a setup with a quality constraint circuit but formally verifies the error constraint by combinational equivalence checking using a satisfiability (SAT) solver. The approach represents a circuit's logic function as AND-inverter graph (AIG) and employs AIG rewriting as approximation technique, i.e., setting nodes to constant zero. Among all possible cuts on the critical paths of the circuit, the one with smallest cut size is selected for rewriting. This heuristic is greedily iterated until there is no more possibility for rewriting without violating the error bound or the maximum number of iterations is reached. For the resulting circuit, area and delay estimates gained by ABC [7] are reported.

Nepal et al. [4] proposed a methodology called ABACUS which transforms a circuit into an Abstract Synthesis Tree (AST). In an iterative approach, transformations on AST are applied to create approximated circuit candidates. The accuracy of the candidates is evaluated by testing, area and power characteristics via ASIC synthesis using a standard cell library. The resulting three metrics are then combined into a fitness function. The candidate with the best fitness is greedily selected as next current circuit. This heuristic process runs for a user-defined number of iterations. Eventually, a Pareto front of designs is given, trading off accuracy for power.

The SCALS framework presented by Liu and Zhang [8] maps an initial gate-level logic network to a target technology. In an iterative process, sub-netlists are extracted from the mapped netlist and are subjected to randomly chosen approximations or optimizations. The candidates are then evaluated by a function including the error, gained through a testing approach, and the area. A Metropolis-Hastings algorithm steers the candidate selection and search until a predefined number of iterations is reached. Additionally, the user can specify a confidence interval for the estimated error. To evaluate the confidence on the estimated error, SCALS employs the T-test.

The ASLAN framework [5] by Ranjan et al. is, to the best of our knowledge, the only presented framework able to approximate sequential circuits while guaranteeing error bounds. In a first step, ASLAN extracts combinational subcircuits amenable to approximation. Then, a search space is generated by creating approximated versions for the subcircuits that vary in their local error constraints and estimated energy consumption. The applied approximation techniques are precision scaling and SALSA [6], although the authors also mention the applicability of other techniques. Finally, ASLAN employs hill climbing to find a locally optimal combination of approximated subcircuits. In each iteration, subcircuit versions with larger error bounds are considered and the combination resulting in the greatest energy savings is selected if the circuit adheres to the global error bound. Otherwise, the next-best combination of subcircuits is picked. Quality assurance relies on a so-called *sequential quality constraint circuit (SQCC)* that raises a flag in case the error bound is violated. Since ASLAN deals with sequential circuits, time frame expansion is used to unroll both the original and the approximated circuit until they finish their computations. The resulting Boolean expression is then formally verified with a SAT solver.

Sengupta et al. proposed SABER [9] which employs two analytical models to estimate the error of arithmetic operations when the bit-width of their input operands are scaled. Using these models, SABER takes the *Data Flow Graph* (DFG) of a given design as input and tries to reduce the bit-width of the operations in a way that the power saving of

[2] http://go.upb.de/paderbench.

Table 1
Overview over presented frameworks for approximate circuit generation.

Category	SASIMI [2]	SALSA [6]	AIG rewriting [3]	ABACUS [4]	SCALS [8]	ASLAN [5]
Circuit type	Combinational	Combinational	Combinational	Combinational + sequential (?)	Combinational	Sequential
Input model	Gate netlist	Gate netlist	Gate netlist/AIG	Behavioral HDL	Gate/LUT netlist	Structural HDL + annotations
Error control	Error bound	Quality function	Error bound	# Iterations	Error bound	Quality evaluation circuit
Search method	Heuristic (hill climbing)	–	Heuristic (greedy)	Heuristic (greedy)	Heuristic (Metropolis-Hastings)	Heuristic (hill climbing)
AC technique	Substitute-and-simplify	Approx. don't care	AIG rewriting	AST transforms	Logic transforms	Precision scaling
Quality assurance	Testing	By construction	Formal verification	Testing	Testing	Formal verification
Output	Approx. circuit	Approx. circuit	Approx. circuit	Pareto front	Approx. circuit	Approx. circuit
Output model	Gate netlist	Gate netlist	Gate netlist (AIG)	Behavioral HDL	Gate/LUT netlist	Structural HDL
Target technology	Standard cell	Standard cell	Techn. independent	Standard cell	Std. cell/LUT-based	Standard cell
Publicly available	–	–	Yes	Yes	–	–

the design is maximized via a heuristic optimizer. Here, the analytical model for estimating the error propagation is limited to just addition and multiplication operations under a restrictive assumption on the distribution of the given design inputs. Indeed, inputs are considered to follow a uniform distribution.

2.2. Classification of existing frameworks

Table 1 presents our attempt to classify the presented frameworks. The challenges are two-fold. Firstly, we need to identify meaningful and orthogonal categories to provide a clear and distinct classification. Secondly, we have to retrieve the required information from related works. Table 1 comprises categories in four groups: the input, the circuit generation/synthesis step, the output, and whether the framework has been made publicly available.

Regarding the input, the first category is the circuit type. Most of the frameworks approximate combinational circuits, while ASLAN was developed for sequential circuits. It has to be noted that it is not necessarily the characteristic of the input circuit that determines its type in our classification. Rather, an approximation technique for sequential circuits means that at least the approximation technique or the quality assurance step (as for ASLAN) are considering the clocked nature of the circuit. For example, several frameworks use sequential circuits such as FIR filters as benchmarks, but restrict the approximation to the datapath and do not report on testing the resulting circuit for a sequence of clock cycles or formally verifying it. Hence we classify these approaches as "combinational" in Table 1. For ABACUS, we are somewhat uncertain how to classify it, since the mentioned approximation techniques are clearly for sequential behavior, e.g., loop transformations, but a corresponding quality assurance was not detailed.

In terms of input model, SASIMI, SALSA, and the approach of [3] rely on gate level netlists. SCALS takes technology-mapped netlists as input, ASLAN begins with circuits described in structural HDL, and ABACUS, operating on a more abstract level, requires behavioral HDL or RTL code. Another issue is how the user can control the error. Most frameworks allow for specifying an error bound, often in several error metrics such as error rate or average error. SALSA and ASLAN define quality functions and quality evaluation circuits, respectively, which encode error bounds. Again, ABACUS is different as it generates a Pareto front showing reasonable trade-offs between accuracy and power. The circuit generation process is controlled by a user-specified number of iterations. Generally, more iterations lead to more approximations and, in turn, can add non-dominated designs to the Pareto front with larger errors.

The second group of rows in Table 1 characterizes the circuit generation step, split into the three categories search, approximation technique, and quality assurance. All techniques rely on heuristics for search. The only exception is SALSA that does not apply a search technique but systematically iterates over the outputs of the approximated circuit to apply don't care optimization. Consequently, SALSA creates circuits that adhere to the error bound by construction, making a subsequent quality assurance step obsolete. ASLAN and the framework in [3] formally verify circuit quality, which is time-consuming but provides a much stronger statement about quality than the testing approaches used in SASIMI, ABACUS, and SCALS. It has to be noted that there are error metrics, e.g., the average-case error [10], for which formal verification cannot be used yet.

The next group of categories characterizes the result produced by the respective framework. Mostly, the tools return one approximated circuit in form of a gate level netlist or structural HDL. ABACUS, however, returns a set of designs in behavioral HDL that form a Pareto front with respect to accuracy and power. Since the results of all frameworks are either netlists or synthesizable hardware descriptions they can potentially target standard cell and FPGA technology. In contrast, with the category "target technology" we refer to the technology used to get estimates for area, delay, and power during the synthesis process. Here, most frameworks target standard cell libraries with the exception of [3], where the AIG representation and ABC functions, respectively, are employed to retrieve technology-independent estimates for area and delay.

Finally, only the authors of ABACUS [4] and the work described in [3] decided to make their frameworks publicly available.

3. Requirements for a flexible framework

Our analysis of related frameworks and the attempt to categorize them has shown that all these approaches have been developed for specific circuit types and limited approximation techniques. In particular, circuit generation is typically described as a monolithic block with interwoven phases for approximation, search, and assuring quality. Moreover, only few frameworks are openly available for experimentation. This situation severely hampers the development and evaluation of new techniques for approximating circuits, and the comparison to existing ones.

With our work, we aim at overcoming these shortcomings and provide a flexible framework for approximate circuit generation. As a starting point for this development, we take our classification of Table 1. This classification provides several categories and shows that many of these are largely orthogonal, giving rise to a reasonable structuring of our framework. Generally, we envision a framework that fulfills the following technical requirements:

- *General:* The framework should not be restricted to certain circuit types, error metrics, approximation and search techniques, or specific target technologies.
- *Modular:* The framework architecture should enable the exchange of

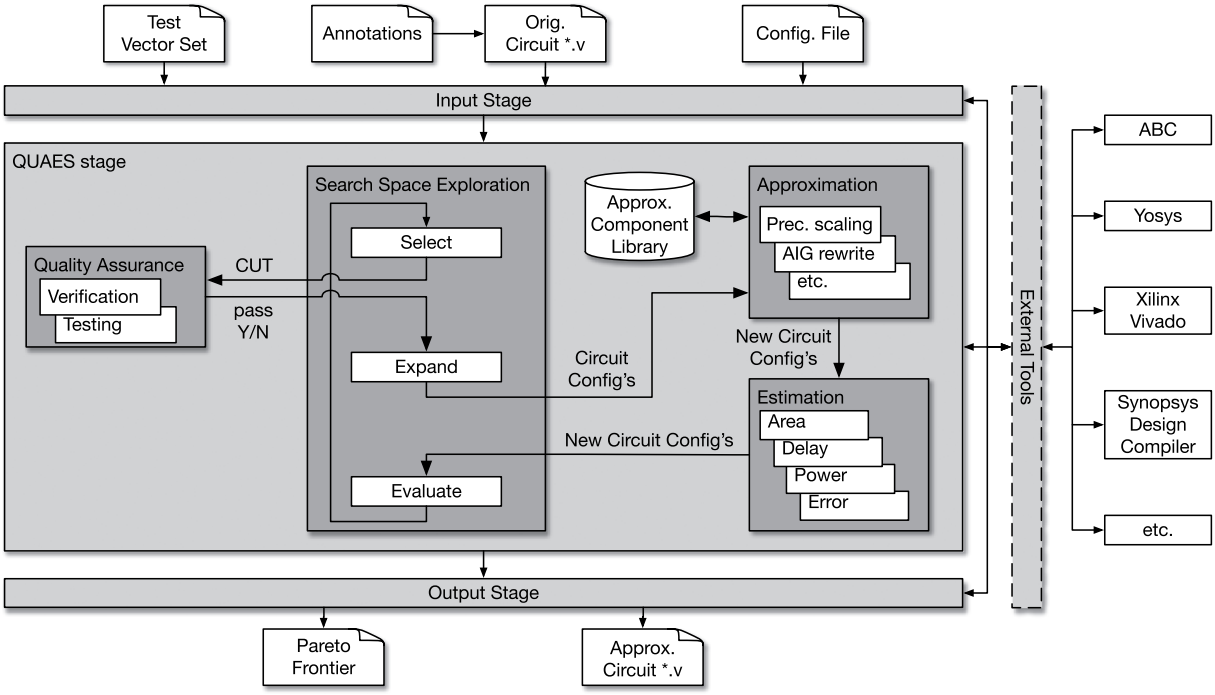


Figure 1. Architecture of the CIRCA approximation framework.

certain processing steps without affecting other steps. Modularity is key for the evaluation and the comparison of different techniques under a consistent experimental setup.

- *Compatible:* The framework, in particular its inputs and outputs, should connect to other, widely-used academic and commercial front-end and back-end tools, e.g., tools synthesizing circuits for ASIC or FPGA technology.
- *Extensible:* The framework should facilitate the swift implementation and evaluation of new techniques.

Additionally, the framework should satisfy the community requirement:

- *Open source availability:* The framework should be publicly available and allow other researchers to use, modify, and extend it. Open-source availability encourages the evaluation and comparison of new techniques against existing ones, and fosters comparative studies.

4. The CIRCA framework

In this section, we present the architecture and main functionality of the CIRCA framework. First, we provide an overview of the framework and then we discuss its stages in more detail.

4.1. Framework overview

Figure 1 shows the architecture of the CIRCA framework. We have developed the architecture with the key features for a flexible approximation framework in mind, namely generality, modularity, extensibility, and compatibility (cmp. Section 3). CIRCA divides into three stages: the input stage, the QUAES stage, and the output stage. The input and output stages frame the QUAES stage and prepare user-supplied circuits for approximation, report on the approximation results, and ensure compatibility between different front-end and back-end tools and the QUAES stage.

The QUAES stage (Quality Assurance, Approximation, Estimation, and Search Space Exploration), the main stage for generating approximate circuits, is designed to implement iterative, search-based approximation approaches with different approximation techniques and both formal verification and testing for circuit validation. Hence, we have organized the QUAES stage into processing blocks, each fulfilling one specific task in the approximation process.

The architecture of CIRCA is kept as general as possible with well-defined interfaces, i.e., data structures representing circuits with annotated information, between its stages and blocks within the QUAES stage, respectively. This allows CIRCA to be easily extended by other techniques and enables the swift exchange of methods to generate a new setup for the approximation process.

4.2. The input stage

The input stage fulfills two main tasks, pre-processing the input design for the approximation process and ensuring compatibility to external formats and tools. In pre-processing, the input stage has to identify a set of subcircuits within the original design which is amenable to approximations, e.g., arithmetic components such as adders and multipliers. This set of subcircuits is denoted in CIRCA as *candidate set*. Suitable candidates can either be identified by a designer through code annotations in the original design (as indicated in Figure 1) or by automated methods. Furthermore, the input stage reads in the user-provided CIRCA configuration file that specifies the functionality of the stages and blocks as well as the test vector set that is required for testing-based quality assurance.

The input stage also provides compatibility between CIRCA's approximation process in the QUAES stage and formats used by external tools, e.g., ABC [7] and Yosys [11]. This ensures that the input design is represented on a level of abstraction and in a format internally processable by CIRCA. Internally supported formats may be, for example, Verilog HDL, Berkeley Logic Interchange Format (BLIF), or Electronic Design Interchange Format (EDIF). Thus, the input stage needs to implement corresponding methods for format conversion.

4.3. The QUAES stage

In the QUAES stage, the candidates are subjected to approximation and

different approximated versions of the candidates will be generated. We denote an approximated version of a candidate as *variant* and the overall circuit with instantiated variants for the candidates as *circuit configuration* or *node*. Different to existing frameworks, CIRCA splits into four independent processing blocks and precisely defines interfaces between them. In this way, CIRCA facilitates the clear distinction between tasks and allows the exchange of methods in one block without affecting the other blocks. The split of the corresponding functionality into the four blocks quality assurance, approximation, estimation, and search space exploration is driven by our classification of related frameworks in Section 2.2 and is key to achieving a modular and extensible architecture.

CIRCA targets search-based approximation processes, i.e., processes which model circuit approximation as a search problem. Iterative search techniques explore the search space by creating and visiting new circuit configurations, which are added as nodes to the search space. Thus, *search space exploration* acts as the central control block of the QUAES stage, invoking the other blocks whenever necessary. The search space exploration block itself comprises the three processing procedures *select*, *expand*, and *evaluate*, procedures that are found in the majority of search algorithms. Relying on these procedures, CIRCA is able to support a wide range of existing search algorithms.

The evaluation step takes a set of circuit configurations and determines estimated values for parameters of interest. Typically, these parameters cover metrics related to area, delay, and power consumption but can also include estimates for the error metrics. The actual estimation functions are encapsulated in estimation procedures to clearly separate estimation from the search space exploration. The estimation procedures will either rely on internal estimation functions or invoke synthesis tools via CIRCA's external tools interface to gather the circuit parameters of interest. The select step receives a set of circuit configurations, each with an annotated vector of estimated parameters, and selects the next configuration to be further considered, i.e., to be expanded. The selection procedure relies on a certain search heuristic. Subsequently, the selected configuration is validated. To this end the circuit configuration, now denoted as *Circuit-under-Test (CUT)*, is sent to the quality assurance procedure. Here, either formal verification or testing is employed to decide if the CUT satisfies the quality constraints and passes the quality assurance check. The quality constraints are provided by the user as inputs to CIRCA and are defined by appropriate error metrics with corresponding error thresholds. If the CUT fails the quality assurance check, the select procedure will, depending on the employed search algorithm, either abort the search or pick the next best circuit configuration for validation. In the latter case, the search terminates if there are no more valid configurations. If the CUT passes validation, the expand procedure grows the search space by calling the approximation block to create new circuit configurations by applying certain approximation techniques. Depending on the actual configuration of the CIRCA framework, one or more candidates of a circuit can be approximated, and one or more approximation techniques can be applied to generate different variants for the candidates.

The approximation block accesses a component library with approximated subcircuits, which is beneficial for two reasons: First, it is rather likely that one circuit component will be approximated for several times. This happens when the overall circuit contains identical candidates, e.g., multiple occurrences of an 8-bit unsigned adder. Storing the approximated versions of such components, i.e., generating a component library on-the-fly, and retrieving them the next time can greatly save computations. Second, CIRCA can leverage already existing libraries of approximated components, e.g., [12,13]. Such libraries could be provided as an input to CIRCA by the user.

4.4. The output stage

The QUAES stage stores all valid nodes, i.e., all circuit configurations that have passed the quality assurance check. The output stage performs post-processing on these circuit configurations. Depending on the CIRCA configuration, the output stage either returns the best approximated circuit, which is the circuit that respects the error constraints and minimizes some user-defined parameter such as area or energy, or provides a Pareto-filtered set of approximated circuits for further analysis and consideration. The output stage also connects to back-end synthesis tools for actual circuit implementation, such as Synopsis Design Compiler for ASIC technology or Xilinx Vivado for FPGA implementation.

5. CIRCA implementation

CIRCA is an open-source software and mainly being coded in Python, following the style guide for Python, PEP-8 [14]. The input stage, the output stage, and the QUAES' processing blocks (quality assurance, approximation, estimation, and search space exploration) are implemented as classes or abstract base classes. During the setup phase, information from the configuration file is either used to set class attributes accordingly or to instantiate the appropriate subclass for the abstract base class. The concept of abstract base classes and abstract methods is used in CIRCA to guide developers through the implementation of new methods and to highlight methods required by CIRCA's approximation flow explicitly.

In this section, we first classify CIRCA into the categories identified in Section 2.2. Then, we present an overview over CIRCA's current implementation by describing the structure of the configuration file and elaborating on the QUAES blocks search space exploration, quality assurance as well as approximation and estimation. Finally, we explain how the user can bound the search space.

5.1. Classification of CIRCA

Table 2 classifies CIRCA into the categories identified in Section 2.2. The table distinguishes between CIRCA's concept and the currently implemented functionality. As the table shows, CIRCA's current implementation provides different methods for the particular categories. Conceptually, however, CIRCA provides large flexibility to developers and a highly configurable system to users.

5.2. The configuration file

Listing 1 shows an exemplary CIRCA configuration file that contains all the information to set up a concrete approximation process. The syntax used is according to Python's *configparser* module, which we employ to process the configuration file. The file is structured into seven configuration sections, labeled with square brackets: General, Input, Quality Assurance, Approximation, Estimation, Search, and Output. The general section defines information relevant to all stages or blocks of the approximation flow, e.g., the name of the design's top module. For the other sections, a *Method* parameter has to be defined. CIRCA uses the method parameter to instantiate the correct class object, providing the desired functionality. Depending on the chosen method, method-dependent parameters might follow, e.g., *TMin* in section *Search*.

During the startup phase of the CIRCA, the configuration file is read by the input stage, the output stage, and the processing blocks of the QUAES stage. Each stage or block, respectively, only extracts the information from its section of the configuration file, i.e., the search processing block only extracts the information provided in the search section of the configuration file. The information provided in the general section is read and processed by all stages or blocks.

5.3. Search space exploration

CIRCA targets iterative, search-based approximation processes. For the synthesis of approximate circuits we typically face very large search spaces that render exhaustive search approaches infeasible. The selection of an appropriate search heuristics or meta-heuristics, respectively, affects the quality of the approximated outcome significantly (see Section 6.3).

Table 2
Overview and classification of CIRCA for approximate circuit generation.

Category	CIRCA (concept)	CIRCA (current implementation)
Circuit type	Combinational + sequential	Combinational + sequential
Input model	Configurable*	Annotated Verilog HDL
Error control	Configurable	Sequential quality constraint circuit[II]
Search method	Configurable	Hill climbing Simulated annealing Monte Carlo Tree Search
AC technique	Configurable	Precision Scaling AIG rewriting
Quality assurance	Configurable	Formal verification
Output	Approx. circuit(s)	Approx. circuit
Output model	Configurable*	Gate netlist
Target technology	Techn. independent	LUT-based
Publicly available	Not applicable	Yes

* For example, SystemC, behavioral or structural HDL, gate netlist, AIG, etc.
II For combinational circuits, registers in the sequential quality constraint circuit are bypassed.

```
[General]
TopModule = Top

[Input]
Method = AnnotatedCandidates # Mandatory
# Method-dependent parameters
Key = <<<APPROX_CAND>>>

[Search]
Method = simulated_annealing # Mandatory
# Method-dependent parameters
TMin = 0.01
Alpha = 0.95
Heuristic = area
Equilibrium = 5

[Estimation]
Method = abc_if # Mandatory

[QualityAssurance]
Method = abc_dprove # Mandatory
# Method-dependent parameters
QualityConstraints = WC:3
CircuitType = streaming
OutputSignal = out
OutputIsSigned = 0

[Approximation]
Method = precision_scaling # Mandatory
QualityConstraints = BF:step=1; # Mandatory

[Output]
Method = output_best_area # Mandatory
```

Listing 1. Exemplary configuration file for CIRCA.

Currently, the CIRCA framework includes the three search methods *hill climbing* (HC), *simulated annealing* (SA), and *Monte Carlo tree search* (MCTS), which are described below. These methods, as the majority of search methods (see Section 4.3), can be divided into three main processing steps: *select*, *expand*, and *evaluate*. In CIRCA, to ensure a correct data flow, these three steps are implemented as abstract methods. The developer only implements these methods when developing a new search algorithm; other processing steps remain unchanged.

5.3.1. Hill climbing(HC)

HC is a greedy method used in CIRCA to minimize a target metric, e.g., energy consumption or hardware area. Our HC method is outlined in Algorithm 1.

Starting from an initial point in the search space — in our case the original circuit *root* — HC iteratively moves in the direction of the most negative difference, i.e., the decisions are only depending on the current state. A drawback for HC, however, is that the search terminates once it reaches a local minimum. Thus, it is not guaranteed to find a global minimum.

For bookkeeping, our HC implementation uses two lists: the *open-list* and the *closed-list*. Nodes or circuit configurations in the open-list can be selected and subjected to further processing, e.g., validation, expansion, or evaluation. Nodes in the closed-list have either already been selected or are showing a positive difference, i.e., are not of interest in the HC search. Initially, the open-list contains only the original circuit *root* while the closed-list is empty.

In the select step, the node with the most negative difference is selected by the function *getBestNode*. Then, the quality assurance block is invoked to validate the node. If the node is invalid, the next best node is selected from the open-list; otherwise, the expand step is employed to expand the currently selected node.

The expand step creates a list of all children[3] and provides this list to the approximation block which then generates the children, i.e., the new circuit configurations. The estimation step is invoked to determine parameters of interest for each node. The evaluation step uses these parameters to compute the difference for each new node in *computeDifferences*. Our current implementation utilizes the difference in the target metric between the currently selected node and the new node. Depending on the user-specified effort level *E*, the open-list is updated. On a low effort level, only the nodes with the most negative difference are considered; on a medium effort level, all nodes with a negative difference are considered; a high effort level considers all nodes with a non-positive difference. If the open-list is empty, the search terminates.

Algorithm 1. Hill climbing algorithm in the CIRCA framework.

Input: O= Verilog description of the circuit E= Effort ϵ_{max}= pre-defined quality threshold
Output: A={Approximated instances satisfying quality constraints}
root ← createRoot(O); // Node with candidate info.
open_list ← {root}; // Nodes for selection
closed_list ← ∅; // Processed nodes

while *!isEmpty(open_list)* **do**
  /* Select step */
  cn ← getBestNode(open_list); // cn: current_node
  closed_list.add(cn);
  open_list.remove(cn);

  /* Validation step */
  cn.valid ← quality_assurance.validate(cn, ϵ_{max});
  if *cn.valid* **then**
    A.add(cn);

    /* Expand step */
    children ← generateChildren(cn);
    /* Only keep children not in open- or closed-list */
    filter(children, open_list);
    filter(children, closed_list);
    open_list.add(children);

    approximation.approximate(children);
    estimation.estimate(children);

    /* Evaluate step */
    computeDifferences(cn, open_list);
    filterDifferencesByEffort(open_ist, E);
  end
end
return A;

[3] If the search space is represented with a tree, we use the terms parent and child instead of neighbor to highlight their relationship.

5.3.2. Simulated annealing

Simulated annealing is a probabilistic search method used to approximate a global optimum under a given runtime constraint. Algorithm 2 shows pseudo-code for CIRCA's SA implementation. First, a random node from the open-list is selected in the select step. Then, *acceptNode* determines the acceptance of the selected node. A node improving the target metric is always accepted for selection. Nodes which worsen the target metric might also be selected to allow for the exploration of the search space. A worse node is accepted under the criteria:

$$random[0, 1) < e^{-\frac{\Delta D}{T}}. \quad (1)$$

T is the current simulated temperature, and ΔD is the change in the target metric, where ΔD is positive for a worse new node. Initially, the simulated temperature T is high, i.e., worse nodes are more likely to be accepted. During the search, however, the temperature cools down at the rate α. The temperature is updated after reaching thermal *equilibrium* ξ at each temperature. In our current implementation we model equilibrium by running the inner loop in Algorithm 2 for a pre-determined number of iterations. A lower temperature leads to a lower probability of worse nodes being selected, i.e., SA moves from exploring the search space at the beginning of the search to moving towards better solutions at the end of the search. Once the temperature T_{Min} is reached, SA terminates.

We assume that increased error constraints lead to larger savings in the target metric hardware area. Thus, in the expand step, only the children of the current node are considered in the open-list which guides the search in the direction of lower quality constraints or higher error constraints, respectively. The search space is pruned by prohibiting moving towards the parent nodes; thus, avoiding a search space explosion.

Algorithm 2. Simulated annealing algorithm in the CIRCA framework.

```
Input: O= Verilog description of the circuit T_Min= Minimum
       temperature α= Cooling rate ξ= Equilibrium ε_max=
       pre-defined quality threshold
Output: A={Approximated instances satisfying quality constraints}
root ← createRoot( O ); // Node with candidate info.
open_list ← {root}; // Nodes for selection
closed_list ← ∅; // Processed nodes
cn ← root;
T ← getInitialTemperature();

while T ≥ T_Min do
    while !EquilibriumReached( ξ ) do
        UpdateEquilibriumState();

        /* Select step                                  */
        if isEmpty( open_list ) then return A ;
        nn ← getRandomNode( open_list );
        closed_list.add( nn );
        open_list.remove( nn );
        if acceptNode( cn, nn, T ) then
            cn ← nn;

            /* Validation step                          */
            cn.valid ← quality_assurance.validate( cn, ε_max );
            if cn.valid then
                A.add( cn );

                /* Expand step                          */
                children ← generateChildren( cn );
                filter( children, closed_list );
                open_list ← children;

                approximation.approximate( children );
                estimation.estimate( children );

                /* Evaluate step                        */
                /* Empty since selection is random
                   */
            end
        end
    end
    t ← t × α; // Update temperature
    ResetEquilibriumState();
end
return A;
```

5.3.3. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a stochastic best-first search technique that exploits the generality of random sampling to build an efficient search algorithm. Recently, it has attracted many interests owing to its success in a wide range of applications [15,16]. MCTS creates an extremely unbalanced search tree towards promising branches through an incremental approach. Indeed, the search effort is allocated to the parts in which the expected achievement is highest in the near future. The current versions of MCTS algorithms, which are widely used in domains such as games, require a slight modification to be fitted for the domain of approximate circuit synthesis [17]. For instance, the simulation time to validate the quality of a circuit instance is very long in comparison with the other domains[4]. This prohibits the possibility of evaluating the current search node by knowing the quality of a subset of probable future nodes originating from here. Consequently, the evaluation of each node is performed independently.

Algorithm 3. MCTS algorithm in CIRCA framework

```
Input: O= Verilog description of the circuit M= Simulation budget
       ε_max = pre-defined quality threshold
Output: A={Approximated instances satisfying quality constraints}

while M > 0 do
    /* Selection step                                   */
    if isEmpty( open_list ) then return A;
    cn ← root;
    while cn.expanded do
        if isEmpty( getActiveChildren( cn )) then
            backpropagate( cn, Nreward ); // Dead end
            break;
        else
            cn ← UCTSelect( getActiveChildren( cn ));
        end
    end
    if !isExpandableNode( cn ) then continue;
    closed_list.add( cn );
    open_list.remove( cn );

    /* Validation step                                  */
    cn.valid ← quality_assurance.validate( cn, ε_max );
    if (cn.valid) then
        A.add( cn );

        /* Expansion step                               */
        children ← generateChildren( cn );
        filter( children, open_list );
        filter( children, closed_list );
        cn.children ← children;
        open_list.add(children);
        cn.expanded ← true;

        approximation.approximate( children );
        estimation.estimate( children );

        /* Evaluation step                              */
        reward ← computeAreaSaving( cn );
        backpropagate( cn, reward );
    else
        cn.active ← false;
        backpropagate( cn, Nreward );
    end
end
return A;
```

The detailed steps of our proposed MCTS are represented in Algorithm 3. We incrementally form a search tree in which a branch represents an approximate transformation and a node specifies an approximate circuit instance, as before. In the *Selection* step the tree is always traversed from the root node, selecting the node with the maximum *Upper Confidence Bound applied to Trees* (UCT) value in each

[4] For example, a playout in Computer-*Go* took just a few micro seconds while the validation of a small size circuit like FIR could take up to a couple of minutes.

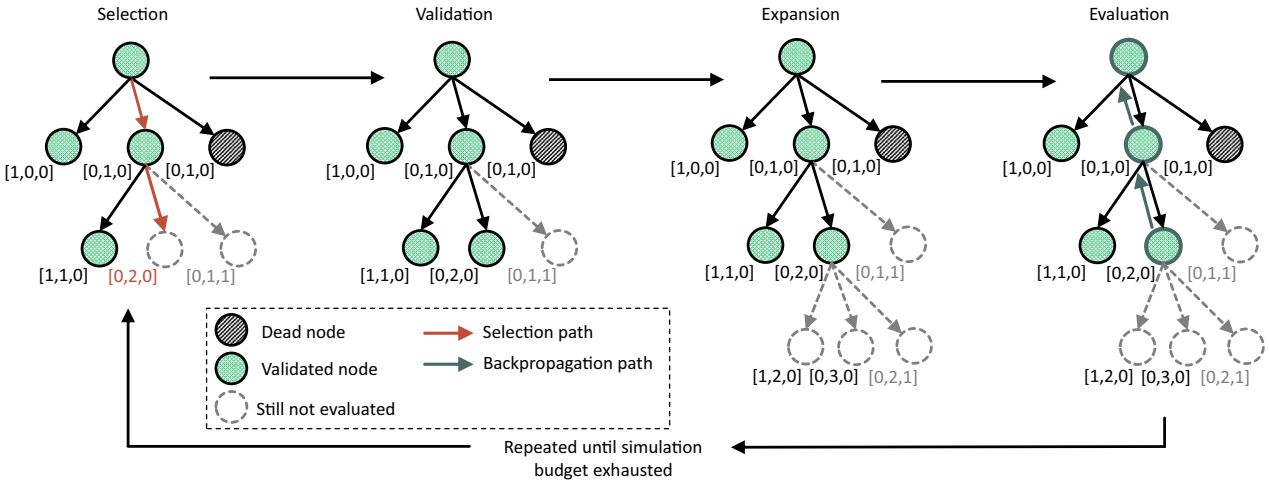


Figure 2. Major steps in the MCTS algorithm.

level. UCT is one of the most common ways to identify the most promising child nodes in the MCTS tree [18], and it is calculated for each node by:

$$UCT(node_i) = \frac{W_i}{V_i} + C\sqrt{\frac{\ln(V_N)}{V_i}} \quad (2)$$

where W_i is the reward value of the $node_i$, and V_i is the number of visits of the $node_i$. Moreover, V_N shows the number of visits for the parent of $node_i$, and C represents the exploration constant which trades off between exploration and exploitation in the search tree, and is practically selected as $C \in [0,2]$ [18]. A benefit of exploiting the UCT policy is to keep a sense of balance in the selection process among nodes that were already evaluated as promising candidates in the previous iterations and the nodes which have not been explored yet. This property makes MCTS suitable to be used in our framework since it is important to have an efficient search technique that explores for potential nodes besides the ones that already proved to be beneficial. To steer subsequent iterations, the algorithm updates the W_i and V_i values of all parent nodes with a *negative reward* using back-propagation if the selected node has no active children left, i.e., is a dead end, thus penalizing the selection path that lead to it.

Next, the quality of the selected node is examined in the *Validation* step to assure that the required precision is not violated. Otherwise, a *negative reward* is again back-propagated from the current node to the root. If the quality constraint is satisfied, the *Expansion* step generates the children of the node. Finally, the reward of the current selected node is calculated in the *Evaluation* step by taking into account the savings in the target metric for the current node. This reward value is then back-propagated up to the root.

Figure 2 illustrates the main steps of the MCTS search algorithm. Here, the root node represents the target circuit that has only three components that can be approximated. Each child represents an approximate circuit configuration with one stride from the root. For example, the child with label [1,0,0] represents a circuit instance with one approximate transformation (e.g., 1-bit LSB truncation) on its first component. In every iteration of MCTS, an approximate candidate is selected and evaluated in term of precision quality. If the quality check is satisfied the node is labeled as a valid node (green) otherwise as a dead node (black striped). Next, the possible children of the current valid node are added to the search tree for future iteration. Then, the benefit of the applied approximation on the current node is calculated as a reward value and back-propagated to the root node. This process iterates until the simulation budget of the search is exhausted. Finally, the best nodes within the tree, i.e., with minimum target metric, are reported.

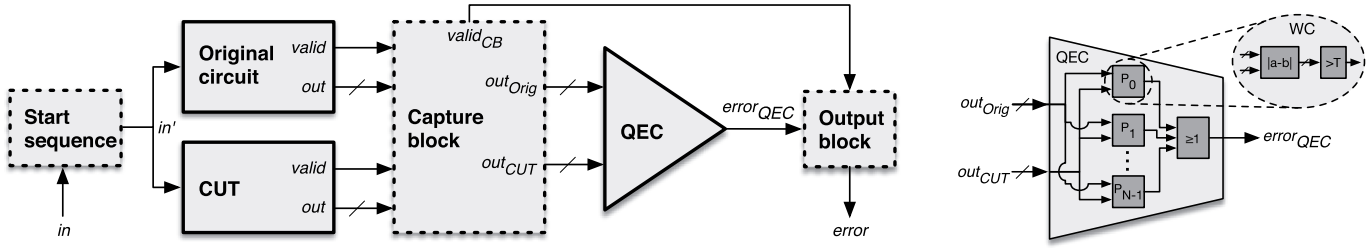
5.4. Quality assurance

For quality assurance, CIRCA can employ testing or formal verification. While approaches based on formal verification lead to conceptually much stronger statements about quality than testing, they also tend to very long runtimes. For testing, a set of test vectors is provided to CIRCA via the input stage. Quality assurance procedures can then apply all these vectors or a randomly selected subset to a circuit-under-test (CUT).

Formal verification focuses on the property *equivalence* between a circuit specification and its implementation. In the context of approximate computing, however, this property needs to be relaxed to *equivalence up to some bound* [19]. For the verification of this property, we use an approximation miter [10] as shown in Figure 3a. Following the terminology of [5], we denote the approximation miter as sequential quality constraint circuit (SQCC).

As Figure 3a outlines, the SQCC comprises of the original circuit, the CUT, and the quality evaluation circuit (QEC) that forms a property checker. The three configurable blocks — depicted by dashed boxes — extend the applicability of our verification approach to a large range of circuits. The block *Start sequence* primes the original circuit and the CUT for operation by implementing a startup protocol. A typical startup protocol might first assert a reset signal and then a start or enable signal, before beginning the actual computation on the input sequence. This additional knowledge may decrease the runtime of the verification since fewer state transitions have to be considered. The *Capture block* and *Output block* work together to ensure that error bounds are checked only at times at which the original circuit and the CUT are specified to generate valid output signals [20].

The QEC generates a single output flag $error_{QEC}$, which is raised if the comparison indicates that the quality constraints are violated. For the specification of quality constraints, CIRCA currently provides the worst-case error and the bit-flip error as error metrics [3]. Note that some error metrics cannot yet be efficiently verified using formal verification methods, e.g., the average-case error [10]. Figure 3b displays the QEC, which holds one or more encoded quality constraints, $P_0,\ldots,P_{N-1}$. The corresponding quality constraint checker modules are OR-ed to form the output flag $error_{QEC}$. The figure furthermore details on the example of P_0, how a worst-case error bound can be encoded. First, the absolute value of the difference of the two outputs of the original circuit and the CUT is computed. Then, the result is compared against a specified threshold T. If all possible deviations are lower than this threshold, the error flag will never be raised, meaning that the quality constraints are never violated. Other error metrics can be encoded similarly. The bit-flip error, for example, is determined by counting the number of differing bits in the two



(a) Sequential quality constraint circuit. (b) Quality evaluation circuit.

Figure 3. Overview of the sequential quality constraint circuit (SQCC) and the quality evaluation circuit (QEC).

output patterns of the original circuit and the CUT.

To formally verify that the quality constraints hold, we need to prove the unsatisfiability of the SQCC's output, i.e., that no sequence of input vectors exists that raises the flag *error* in some clock cycle. CIRCA currently offers two different inductive solvers for this step, which are the tools included with ABC [7] and Yosys [11].

5.5. Approximation and estimation

Generally, the approximation block receives a set of circuit configurations with candidates indicated for approximations. CIRCA allows multiple approximation techniques to be employed during the approximation process. Currently, two approximation techniques are implemented: precision scaling (PS) and approximation-aware AIG rewriting (AIG) [3]. Both AIG, carefully following the descriptions in [3], and PS have been implemented into the widely-used ABC tool [7] to have both techniques available in a tool which provides a broad range of synthesis, optimization, and verification functionality.

The circuit configurations are then passed on to the estimation block and, at the same time, the approximated variants for the candidates are stored in a library of approximated components for later re-use (see Section 4.3). The estimation block receives a list of circuit configurations with newly approximated component variants and estimated target metrics of interest. Currently, CIRCA uses ABC's *if* command to estimate area parameters for FPGA 4-LUT architectures. The parameters are stored in the circuit object and can be used for further processing.

5.6. Bounding the search space

To allow for a search space expansion in a controlled manner, CIRCA differentiates between global and *local quality constraints*. Global quality constraints apply to the circuit configurations and have been described in Section 5.4. Local quality constraints are constraints on error metrics and apply to generated variants of circuit components, e.g., the worst-case error of the variant can be bounded. In CIRCA this happens in the approximation block. As a result, in the course of the approximation process the variants created for one candidate will exhibit a step-wise decrease in accuracy. Naturally, these local quality constraints do not directly relate to the overall circuit's quality, which is checked in the quality assurance block.

The user can bound the search space by controlling its density and dimensionality. The density is given by the distance between directly neighboring nodes in the search space. CIRCA allows the user to determine the density by setting step sizes for the local quality constraints. For example, a step size of 0.1% for the local error metric worst-case error will populate the search space with doubled density compared to a step size of 0.2% Furthermore, in each iteration of the search process CIRCA approximates n candidates $C_0,\ldots,C_{n-1}$ of a circuit configuration by applying to each candidate C_i a number of A_{C_i} different approximation techniques and checking for E_{C_i} different local error metrics. These parameters span the dimensions of the search problem. Overall, the total number of dimensions D of the search space is given by Eq. (3).

$$D = \sum_{i=0}^{n-1} A_{C_i} \times E_{C_i} \tag{3}$$

By setting appropriate values for n, A_{C_i} and E_{C_i} the user limits the complexity of the search problem. For example, for the experiments presented in Section 6, we use $A_{C_i} = E_{C_i} = 1 \quad \forall\, i = 1...n$ with n ranging between 2 and 23, as denoted in Table 3. For readability, the representation of the dimensions and densities has been excluded from the descriptions of the search techniques in Section 5.3.

6. Experiments

In this section, we present a set of exemplary experiments to demonstrate the capabilities of CIRCA. We first provide information about the experimental setup, followed by a description of the used benchmark circuits and, finally, we show and discuss the actual results of our experimentation runs.

6.1. Experimental setup

For experimentation, we have selected seven sequential circuits from our benchmark suite PaderBench and manually annotated adder and multiplier components in the data path as approximation candidates. CIRCA has been set up to vary the worst-case error bound from 0.25% to 5.0%, expressed in percentage of the circuit's maximum possible output value, to employ formal verification with ABC's *dprove* for assuring quality and the hardware area as target metric. Moreover, we have systematically experimented with all three search methods HC, SA, and MCTS in different parametrization and with precision scaling AND-inverter graph rewriting as approximation techniques.

We have run the approximation flow five times for each benchmark circuit and determined the averages as representative results. The experiments have been performed on a compute cluster which runs Scientific Linux 7.2 (Nitrogen), comprises nodes with an Intel® Xeon E5-2670@2.6 GHz (16 cores) and from 64 up to 256 Gigabyte main memory, of which it provides 2 Gigabyte main memory and one core per job, i.e., single Benchmark run.

6.2. Benchmark suite PaderBench

We have compiled a benchmark suite named PaderBench to facilitate quick prototyping with CIRCA. PaderBench has been made publicly available and each benchmark circuit is accompanied by input test vectors so the results may be reproduced when a testing-based approach is used to validate the circuit's quality. A subset of the benchmarks of PaderBench has been selected for conducting the experiments described in this paper. Table 3 elaborates on the characteristics of these benchmark circuits, which are the output bit widths, the hardware area in terms of number of used FPGA 4-LUTS as reported by ABC [7], and the number of manually identified candidates for approximation. The selected sequential circuits represent various application domains ranging from small arithmetic blocks to complex

Table 3
Sequential benchmark circuits.

Circuit Name	Description	Output Bit Width	#4-LUTs[§]	#Candidates
butterfly [21]	Operation used in FFT	32[†]	19038	7
fir_8tap	FIR filter 8-tap	67	12401	15
fir_pipe_16 [22]	FIR filter 16-tap	18	8935	23
pipeline_add [23]	Pipelined adder	40	572	2
rgb2ycbcr [24]	Color-space transformation	24[‡]	4981	5
ternary_sum_nine [23]	Adder tree	20	1484	4
weight_calculator	Industrial scale	12	2272	4

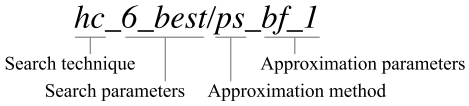
[§] ABC estimation after mapping.
[†] Concatenation of real and imaginary part.
[‡] Concatenation of three channels, each 8-bit wide.

datapaths and are briefly described as follows:

- *butterfly* is the basic operation in the FFT/DFT, which is used in many signal processing applications. It takes two 32-bit wide complex numbers and a 32-bit twiddle factor as input and returns the result of computation as a 32-bit complex number [21].
- *fir_8tap* implements an eight-tap Gaussian low-pass FIR filter, which is commonly used in the domain of signal processing. It takes a 32-bit wide data input and provides a 67-bit wide output.
- *fir_pipe_16* is a 16-tap FIR filter [22] that uses pipelined blocks to increase the throughput. Both the input and output are 18-bit values.
- *pipeline_add* [23] is a pipelined adder circuit that takes two 20-bit numbers as input and gives a 20-bit sum as output.
- *rgb2ycbcr* is a module in the JPEG encoder [24] that performs a color space transformation. It takes a 24-bit RGB value as input and provides a 24-bit wide output which represents the YCbCr color space value.
- *ternary_sum_nine* is an adder tree [23] that performs an addition of nine binary words using four adder chains. Its input comprises nine values to be added, each 16-bit wide, and the output is a 20-bit value containing the sum.
- *weight_calculator* is a simplified version of an industrial scale weight calculator which is used to control the hoppers of a multi-head weigher. It reads an input value stored in RAM and provides outputs to control the hoppers. For verification purposes a 12-bit output that represents the combined weight of the hoppers is used in our experiments.

6.3. Experimental results

In this paper, we use a concise and consistent denotation to differentiate between experiments. The denotation comprises of two parts separated by a *slash (/)*. The part before the slash defines the setup of the search procedure, the second part after the slash describes the approximation techniques. Both parts contain a number of parameters separated by *underscores (_)*:



For example, the above denotation describes a CIRCA setup with hill climbing as search technique, parametrized with *6_best*, and precision scaling as approximation method, parametrized with *bf_1*. For hill climbing search, the first part of the denotation assembles as follows: *hc_effort_selection-strategy*. The *effort* level ranges from 0 to 6, where effort level 0 means that only nodes with the most negative difference are considered, effort level 5 considers all nodes with a negative difference, and level 6 considers all nodes with non-positive difference. As *selection strategy*, we take the *best* node, i.e., a node with the most negative difference, which goes together with effort level 0. An alternative selection strategy could, for example, select a random node from the list of considered nodes.

The first part of the denotation for simulated annealing (SA) is structured as *sa_T_{Min}_alpha_equilibirum*. For all our SA experiments, we have used 1.0 as initial temperature. The parameters *alpha*, T_{Min} and *equilibirum* define the SA schedule as detailed in Section 5.3.2 (cooling rate, temperature target, inner loop iterations). *Alpha* and T_{Min} denote the corresponding internal decimal places only.

The first part of the denotation for Monte Carlo Tree Search (MCTS) includes two parameters, i.e., *mcts_budget_exploration-constant*. The *budget* determines the maximum number of iterations for MCTS. The *exploration constant* matches the exploration constant described in Section 5.3.3, scaled by a factor of $\frac{1}{10}$.

The approximation part of our denotation composes of *approx._technique_local-quality-metric_step-size*, where the employed approximation technique is either *ps* for precision scaling or *aig* for approximation-aware AIG rewriting. The *bf* (bit-flip error) or *wc* (worst-case error) are used as local quality metrics for the candidates or variants, respectively. The *step-size* specifies the distance between neighboring variants.

The experimental results are presented in Figure 4. The figure displays the area normalized to the area of the original circuit over the worst-case error bound. The figure divides into seven bar plots, one for each benchmark. Each bar plot shows results achieved with different setups regarding the search method and approximation technique.

Comparing the approximation techniques over the experiments, we see that PS could achieve area savings of up to ≈55%, while AIG could only achieve area savings of up to ≈25%. An explanation for the superiority of PS over AIG in our experiments is that we have selected arithmetic components as candidates. For such components, PS degrades the accuracy more gracefully than AIG. AIG targets nodes on the critical path, which usually affects the carry-chain of a multiplier or adder. Approximating such nodes leads to large errors and, in turn, to the rejection of the variant if moderate error bounds are applied. PS introduces smaller errors when operating on the LSBs of the output vector of a candidate and thus approaches the error boundaries more carefully.

Figure 4 also allows to study the impact of the employed search method on the quality of the approximated outcome. Differences in the results can be observed among the same search methods when parametrized differently, and also among the different search methods. When evaluating HC, it can be seen that, in general, higher effort levels lead to higher area savings. For SA, however, increasing the number of allowed iterations does not lead to significant differences in area savings. On the one hand, this could be caused by our implementation's method for pruning off the search space. On the other hand, fine-tuning the search parameter might lead to further improvements. However, in general SA performs very well and achieves higher savings than HC — especially, for the benchmarks pipeline_add and rgb2ycbcr. MCTS performance lies between HC and SA. For an improved performance,

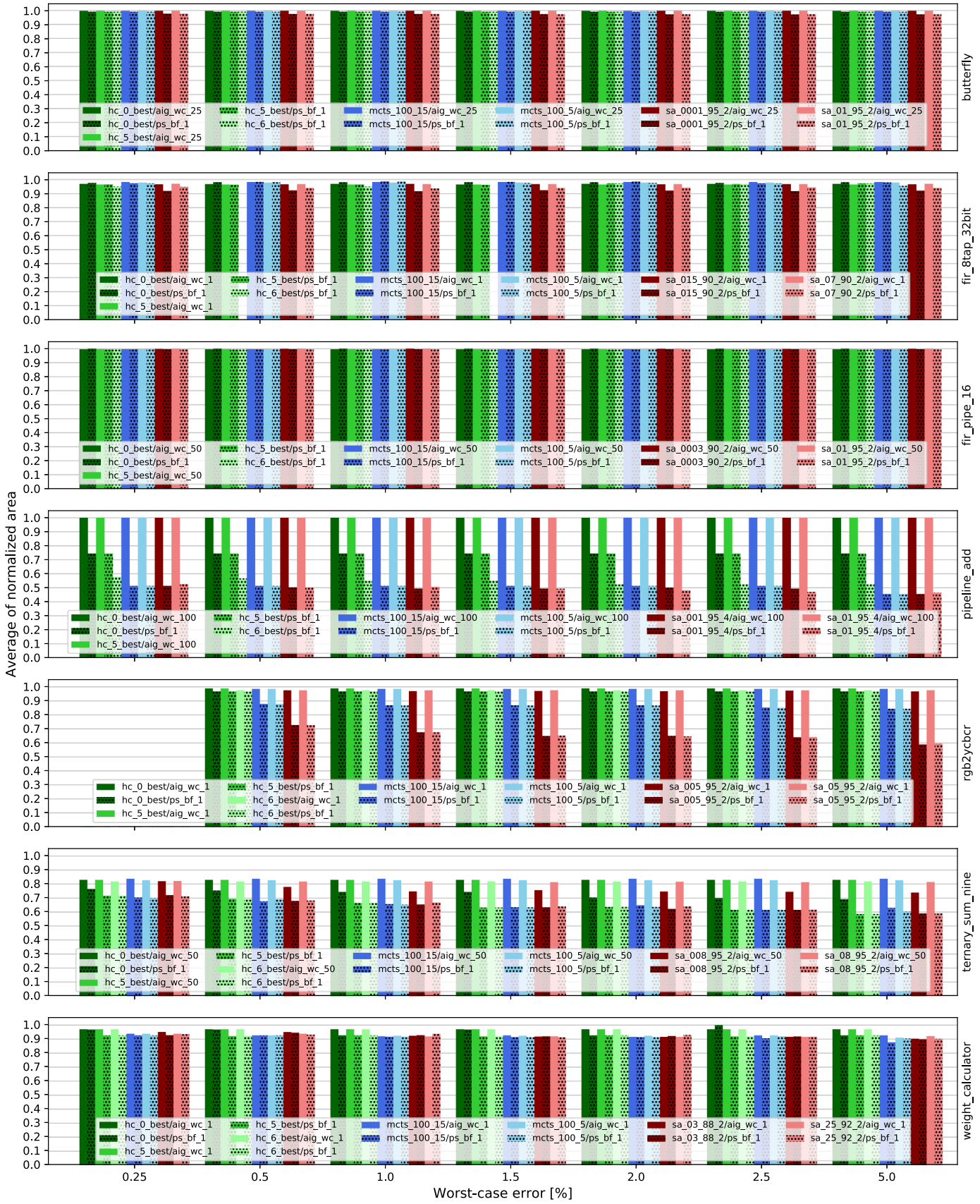


Figure 4. Average of normalized area of approximated circuits generated by CIRCA.

Table 4
Average runtimes of the CIRCA framework and number of selected nodes.

Identifier	Worst-case error bound [%] 0.25		0.50		1.0		1.50		2.00		2.50		5.00	
butterfly														
hc_0_best/aig_wc_25	1:02:21	(4)	52:47	(4)	50:24	(4)	57:58	(4)	45:47	(4)	34:08	(4)	42:34	(4)
hc_0_best/ps_bf_1	40:28	(4)	45:16	(4)	49:10	(4)	1:00:21	(4)	49:49	(4)	39:46	(4)	46:12	(4)
hc_5_best/aig_wc_25	1:04:30	(8)	1:02:06	(8)	1:02:16	(8)	1:00:44	(8)	1:21:47	(8)	1:37:15	(8)	1:56:42	(8)
hc_5_best/ps_bf_1	54:46	(9)	1:05:10	(9)	54:02	(9)	1:06:53	(9)	54:40	(9)	1:03:16	(9)	1:12:04	(9)
hc_6_best/aig_wc_25	—†	—	—†	—	—†	—	—†	—	—†	—	—†	—	—†	—
hc_6_best/ps_bf_1	1:11:16	(9)	1:24:54	(9)	1:11:34	(9)	1:08:27	(9)	1:45:38	(9)	1:12:43	(9)	1:20:03	(9)
mcts_100_15/aig_wc_25	4:36:20	(94)	4:26:59	(94)	4:30:33	(94)	4:31:09	(94)	4:28:05	(94)	4:27:34	(94)	4:26:58	(94)
mcts_100_15/ps_bf_1	4:00:26	(95)	3:50:56	(95)	3:50:41	(95)	3:48:28	(95)	3:49:04	(95)	3:50:12	(95)	3:54:07	(95)
mcts_100_5/aig_wc_25	4:55:09	(100)	4:54:59	(100)	4:48:04	(100)	4:50:25	(100)	4:50:06	(100)	4:53:50	(100)	4:50:29	(100)
mcts_100_5/ps_bf_1	3:56:23	(95)	3:55:04	(95)	3:47:40	(95)	3:47:25	(95)	3:51:50	(95)	3:54:50	(95)	3:50:07	(95)
sa_0001_95_2/aig_wc_25	1:08:26:07	(360)	1:06:55:45	(360)	1:08:15:43	(360)	1:08:13:53	(360)	1:06:00:24	(360)	1:06:10:33	(360)	1:06:25:14	(360)
sa_0001_95_2/ps_bf_1	6:56:40	(105)	9:01:36	(112)	7:34:08	(120)	7:35:18	(115)	9:53:01	(117)	9:41:53	(112)	13:35:25	(125)
sa_01_95_2/aig_wc_25	15:23:08	(180)	13:32:12	(180)	11:44:22	(180)	13:33:15	(180)	13:34:58	(180)	13:45:41	(180)	15:15:19	(180)
sa_01_95_2/ps_bf_1	9:19:15	(117)	11:12:22	(114)	9:29:05	(120)	9:02:28	(114)	9:38:40	(120)	7:24:04	(116)	7:30:44	(122)
fir_8tap_32bit														
hc_0_best/aig_wc_1	1:34:51	(7)	1:38:54	(8)	1:25:54	(7)	1:51:05	(9)	1:22:59	(7)	1:46:13	(7)	1:33:10	(7)
hc_0_best/ps_bf_1	9:25:14	(5)	7:33:10	(4)	7:49:28	(4)	7:32:53	(4)	7:32:53	(4)	8:33:10	(5)	7:33:13	(4)
hc_5_best/aig_wc_1	5:06:09	(22)	4:22:32	(22)	4:21:33	(22)	4:19:26	(22)	4:17:12	(22)	5:26:55	(22)	4:21:40	(22)
hc_5_best/ps_bf_1	8:22:18:02	(44)	11:11:03:29	(59)	8:22:33:25	(43)	12:16:51:15	(64)	3:08:52:30	(25)	8:10:27:19	(41)	3:14:55:14	(25)
hc_6_best/aig_wc_1	—†	—	—†	—	—†	—	—†	—	—†	—	—†	—	—†	—
hc_6_best/ps_bf_1	9:16:58:40	(74)	—†	—	11:09:44:17	(67)	—†	—	3:23:08:46	(28)	10:20:12:24	(50)	4:04:41:12	(28)
mcts_100_15/aig_wc_1	4:49:24	(100)	4:55:51	(100)	4:53:22	(100)	4:52:37	(100)	4:54:48	(100)	4:52:19	(100)	5:01:21	(100)
mcts_100_15/ps_bf_1	8:01:05:39	(100)	9:14:51:55	(100)	7:11:40:04	(100)	7:23:22:51	(100)	7:09:20:37	(100)	8:11:59:16	(100)	2:00:51:45	(100)
mcts_100_5/aig_wc_1	5:05:56	(100)	4:54:07	(100)	5:03:47	(100)	5:04:42	(100)	5:04:21	(100)	5:05:25	(100)	4:59:22	(100)
mcts_100_5/ps_bf_1	8:15:08:01	(100)	6:17:50:00	(100)	7:15:14:11	(100)	1:11:46:04	(100)	23:36:19	(100)	8:11:12:37	(100)	16:27:45	(100)
sa_015_90_2/aig_wc_1	9:40:57	(80)	11:37:48	(80)	14:04:00	(80)	13:56:32	(80)	12:01:49	(80)	11:47:47	(80)	13:51:39	(80)
sa_015_90_2/ps_bf_1	2:17:31:50	(80)	3:14:04:07	(80)	1:18:23:51	(80)	3:12:22:52	(80)	2:03:55:05	(80)	2:04:25:36	(80)	2:07:08:38	(80)
sa_07_90_2/aig_wc_1	5:23:12	(52)	9:15:08	(52)	7:28:21	(52)	7:26:53	(52)	7:35:21	(52)	11:42:42	(52)	7:35:41	(52)
sa_07_90_2/ps_bf_1	4:18:18:55	(52)	1:22:00:47	(52)	2:04:56:04	(52)	2:16:28:32	(52)	2:02:40:02	(52)	3:03:25:52	(52)	1:02:06:39	(52)
fir_pipe_16														
hc_0_best/aig_wc_50	1:11:07	(4)	1:40:46	(4)	1:39:15	(4)	51:12	(4)	1:47:36	(4)	51:34	(4)	1:13:00	(4)
hc_0_best/ps_bf_1	45:39	(2)	55:04	(2)	59:15	(2)	42:41	(2)	30:47	(2)	40:10	(2)	41:36	(2)
hc_5_best/aig_wc_50	2:52:10	(7)	2:44:43	(7)	2:43:36	(7)	2:51:59	(7)	3:44:51	(7)	2:42:33	(7)	1:14:19	(7)
hc_5_best/ps_bf_1	3:54:01	(66)	4:41:54	(66)	4:53:26	(66)	4:09:07	(66)	5:02:07	(66)	4:10:31	(66)	5:00:36	(66)
hc_6_best/aig_wc_50	—†	—	—†	—	—†	—	—†	—	—†	—	—†	—	—†	—
hc_6_best/ps_bf_1	10:40:48	(264)	9:57:15	(264)	11:43:27	(264)	10:47:35	(264)	13:12:34	(264)	12:50:07	(264)	11:51:58	(264)
mcts_100_15/aig_wc_50	7:59:31	(100)	7:03:34	(100)	7:32:02	(100)	8:20:24	(100)	8:00:53	(100)	7:50:02	(100)	7:52:15	(100)
mcts_100_15/ps_bf_1	3:34:35	(100)	3:41:17	(100)	3:37:26	(100)	3:35:20	(100)	3:53:30	(100)	3:33:54	(100)	3:42:44	(100)
mcts_100_5/aig_wc_50	8:29:07	(100)	8:44:01	(100)	7:31:55	(100)	7:13:30	(100)	7:30:39	(100)	7:36:19	(100)	7:19:50	(100)
mcts_100_5/ps_bf_1	3:36:11	(100)	3:33:58	(100)	3:32:56	(100)	3:34:30	(100)	3:41:31	(100)	3:35:29	(100)	3:35:06	(100)
sa_0003_90_2/aig_wc_50	21:03:42	(154)	1:01:12:54	(154)	22:47:51	(154)	22:49:41	(154)	22:38:35	(154)	20:43:53	(154)	1:00:43:35	(154)
sa_0003_90_2/ps_bf_1	9:48:41	(106)	8:15:25	(108)	8:33:44	(114)	12:16:37	(118)	12:27:52	(120)	10:01:33	(111)	10:17:55	(111)
sa_01_95_2/aig_wc_50	1:01:02:11	(180)	1:00:30:35	(180)	1:02:59:09	(180)	1:02:46:13	(180)	1:03:18:13	(180)	22:37:33	(180)	1:00:32:46	(180)
sa_01_95_2/ps_bf_1	15:08:52	(177)	13:41:05	(180)	17:06:49	(179)	15:04:52	(179)	17:13:22	(180)	15:25:17	(179)	15:03:33	(180)
pipeline_add														
hc_0_best/aig_wc_100	7	(1)	7	(1)	7	(1)	53	(1)	7	(1)	7	(1)	1:03	(1)
hc_0_best/ps_bf_1	14:13	(16)	1:12	(16)	7:02	(16)	1:13	(16)	1:13	(16)	1:12	(16)	1:11	(16)
hc_5_best/aig_wc_100	1:03	(1)	1:04	(1)	1:03	(1)	1:03	(1)	2:10	(1)	2:42	(1)	2:40	(1)
hc_5_best/ps_bf_1	6:37	(16)	6:32	(16)	6:45	(16)	6:37	(16)	6:39	(16)	6:39	(16)	9:36	(16)
hc_6_best/aig_wc_100	—†	—	—†	—	—†	—	—†	—	—†	—	—†	—	—†	—
hc_6_best/ps_bf_1	31:02	(29)	22:31	(30)	23:42	(31)	22:31	(31)	22:45	(31)	23:42	(31)	30:15	(31)
mcts_100_15/aig_wc_100	2:43	(41)	2:35	(41)	2:46	(41)	2:34	(41)	2:42	(41)	2:34	(41)	3:01	(41)
mcts_100_15/ps_bf_1	6:11	(89)	6:19	(92)	6:28	(94)	6:32	(94)	6:00	(89)	5:58	(89)	5:36	(83)
mcts_100_5/aig_wc_100	2:29	(41)	2:19	(41)	2:34	(41)	2:26	(41)	2:44	(41)	2:32	(41)	2:47	(41)
mcts_100_5/ps_bf_1	5:19	(79)	5:15	(78)	5:32	(82)	5:57	(82)	5:15	(87)	5:17	(87)	5:18	(86)
sa_001_95_4/aig_wc_100	6:44:17	(540)	5:03:45	(540)	5:12:44	(540)	6:45:50	(540)	6:55:48	(540)	4:53:54	(540)	3:18:25	(540)
sa_001_95_4/ps_bf_1	14:21	(42)	33:31	(40)	35:45	(38)	15:29	(40)	19:14	(38)	19:16	(38)	22:04	(41)
sa_01_95_4/aig_wc_100	29:18	(360)	29:32	(360)	4:00:49	(360)	3:38:11	(360)	3:46:23	(360)	3:34:51	(360)	4:07:19	(360)
sa_01_95_4/ps_bf_1	1:03:14	(38)	40:40	(39)	48:07	(37)	35:19	(41)	27:53	(40)	31:41	(40)	6:51	(39)

(*continued on next page*)

Table 4 (*continued*)

Identifier	Worst-case error bound [%] 0.25		0.50		1.0		1.50		2.00		2.50		5.00	
rgb2ycbcr[§]														
hc_0_best/aig_wc_1			1:46	(2)	1:47	(2)	1:50	(2)	6:16	(2)	4:34	(2)	1:49	(2)
hc_0_best/ps_bf_1			15:03	(6)	41:24	(6)	1:17:45	(6)	24:29	(6)	1:14:19	(6)	28:21	(6)
hc_5_best/aig_wc_1			16:09	(3)	16:15	(3)	16:32	(3)	16:08	(3)	11:08	(3)	10:48	(3)
hc_5_best/ps_bf_1			22:07	(11)	59:27	(11)	1:50:57	(11)	1:04:56	(11)	1:48:45	(11)	55:15	(11)
hc_6_best/aig_wc_1			2:53:07	(52)	3:06:47	(52)	2:22:43	(52)	3:59:46	(52)	3:51:37	(52)	3:18:13	(52)
hc_6_best/ps_bf_1			36:24	(11)	1:10:54	(11)	1:37:13	(11)	54:40	(11)	1:32:53	(11)	52:48	(11)
mcts_100_15/aig_wc_1			2:13:00	(97)	1:44:16	(97)	1:37:42	(97)	1:50:15	(97)	1:44:30	(97)	1:28:23	(97)
mcts_100_15/ps_bf_1			3:05:08	(100)	3:53:01	(100)	3:52:15	(100)	4:06:43	(100)	4:42:26	(100)	10:20:47	(100)
mcts_100_5/aig_wc_1			2:02:57	(96)	1:34:52	(96)	1:34:19	(96)	1:37:29	(96)	1:28:40	(96)	1:22:38	(96)
mcts_100_5/ps_bf_1			3:07:06	(100)	3:23:03	(100)	3:52:08	(100)	3:48:55	(100)	4:38:44	(100)	9:49:55	(100)
sa_005_95_2/aig_wc_1			6:26:36	(208)	7:44:05	(208)	7:42:56	(208)	10:37:17	(208)	8:21:10	(208)	18:50:54	(208)
sa_005_95_2/ps_bf_1			2:39:54	(85)	2:44:46	(88)	4:18:43	(94)	3:20:04	(94)	5:07:03	(93)	6:51:58	(101)
sa_05_95_2/aig_wc_1			2:57:32	(118)	5:58:38	(118)	3:59:37	(118)	6:24:48	(118)	6:27:43	(118)	10:06:10	(118)
sa_05_95_2/ps_bf_1			1:09:26	(84)	1:29:10	(88)	2:41:00	(93)	2:59:23	(95)	1:38:51	(94)	4:24:12	(99)
ternary_sum_nine														
hc_0_best/aig_wc_50	22:14	(14)	19:54	(14)	27:46	(14)	13:50	(14)	13:09	(14)	32:35	(14)	23:24	(14)
hc_0_best/ps_bf_1	21:46	(21)	3:06	(22)	24:06	(23)	19:16	(23)	25:15	(26)	9:32	(27)	16:22	(28)
hc_5_best/aig_wc_50	59:57	(22)	54:35	(22)	1:00:10	(22)	45:54	(22)	37:18	(22)	36:05	(22)	42:17	(22)
hc_5_best/ps_bf_1	37:40	(72)	1:48:21	(80)	39:43	(85)	1:08:33	(99)	40:50	(93)	1:08:27	(101)	1:34:12	(140)
hc_6_best/aig_wc_50	1:07:51	(24)	45:52	(24)	52:22	(24)	48:08	(24)	53:41	(24)	40:50	(24)	39:01	(24)
hc_6_best/ps_bf_1	1:40:19	(72)	1:25:23	(92)	1:27:37	(93)	42:40	(99)	1:27:04	(93)	1:10:40	(113)	1:28:57	(140)
mcts_100_15/aig_wc_50	2:05:21	(99)	1:24:25	(99)	1:00:10	(99)	51:28	(99)	42:27	(99)	41:36	(99)	27:40	(99)
mcts_100_15/ps_bf_1	10:32	(100)	10:38	(100)	10:39	(100)	10:33	(100)	10:48	(100)	10:34	(100)	10:50	(100)
mcts_100_5/aig_wc_50	2:10:16	(94)	1:16:23	(94)	56:42	(94)	50:07	(94)	38:01	(94)	38:42	(94)	26:17	(94)
mcts_100_5/ps_bf_1	11:40	(100)	10:44	(100)	10:41	(100)	10:58	(100)	10:53	(100)	11:04	(100)	10:53	(100)
sa_008_95_2/aig_wc_50	5:01:04	(131)	8:23:49	(168)	3:18:42	(190)	6:15:05	(190)	4:01:09	(190)	4:22:21	(189)	2:44:18	(190)
sa_008_95_2/ps_bf_1	1:39:01	(43)	1:02:59	(50)	1:15:10	(52)	1:21:10	(56)	1:53:40	(62)	1:18:14	(63)	1:21:21	(64)
sa_08_95_2/aig_wc_50	3:05:57	(97)	2:23:13	(100)	3:02:48	(100)	4:12:25	(100)	2:39:03	(100)	3:28:07	(100)	3:20:30	(100)
sa_08_95_2/ps_bf_1	18:48	(45)	19:52	(51)	6:50	(50)	38:23	(58)	23:07	(54)	37:28	(62)	47:41	(63)
weight_calculator														
hc_0_best/aig_wc_1	13:11	(2)	10:14	(2)	12:55	(2)	17:30	(2)	10:09	(2)	10:16	(2)	13:23	(2)
hc_0_best/ps_bf_1	7:20:39	(3)	7:22:14	(3)	31:23	(3)	10:51:38	(3)	24:49	(3)	7:10:31	(2)	21:34	(3)
hc_5_best/aig_wc_1	13:56	(2)	13:02	(2)	13:37	(2)	12:23	(2)	12:32	(2)	12:05	(2)	5:58	(2)
hc_5_best/ps_bf_1	15:59:27	(10)	1:12:23:30	(11)	35:56	(3)	11:42:54	(8)	25:04	(3)	23:14:32	(9)	22:00	(3)
hc_6_best/aig_wc_1	4:04	(2)	6:13	(2)	4:05	(2)	13:40	(2)	10:13	(2)	10:46	(2)	10:59	(2)
hc_6_best/ps_bf_1	16:12:20	(13)	1:12:19:50	(11)	29:55	(3)	11:55:09	(8)	31:10	(3)	23:14:20	(9)	17:03	(3)
mcts_100_15/aig_wc_1	5:18:47:44	(92)	5:10:43:45	(65)	2:19:57:28	(98)	3:10:49:07	(99)	1:22:20:06	(100)	2:22:10:17	(100)	14:54:21	(98)
mcts_100_15/ps_bf_1	2:04:29:29	(42)	2:01:19:13	(33)	7:06:41:36	(92)	4:21:43:29	(84)	4:13:38:34	(91)	4:20:36:04	(99)	4:13:20:58	(99)
mcts_100_5/aig_wc_1	5:11:36:44	(92)	4:22:39:15	(58)	3:11:58:53	(99)	2:13:49:07	(99)	1:22:32:26	(100)	2:21:45:30	(100)	2:02:32:00	(100)
mcts_100_5/ps_bf_1	2:04:01:37	(42)	2:01:22:18	(33)	7:14:21:44	(91)	5:10:54:56	(86)	5:06:31:57	(92)	1:20:22:53	(51)	4:16:56:27	(99)
sa_03_88_2/aig_wc_1	18:32:05	(14)	2:17:54:56	(22)	2:16:34:13	(27)	3:03:36:21	(35)	3:02:10:29	(36)	2:17:47:01	(43)	2:07:58:38	(56)
sa_03_88_2/ps_bf_1	14:32:53	(9)	22:32:01	(10)	1:17:41:37	(15)	1:04:02:21	(14)	1:11:46:32	(19)	1:18:41:58	(18)	1:02:36:46	(20)
sa_25_92_2/aig_wc_1	14:11:12	(14)	1:22:11:05	(17)	3:07:05:40	(32)	2:13:52:05	(33)	2:06:51:14	(34)	1:14:49:32	(34)	1:06:34:56	(33)
sa_25_92_2/ps_bf_1	8:39:01	(8)	22:51:42	(9)	1:13:08:49	(13)	1:18:00:59	(16)	1:04:53:25	(16)	1:09:53:11	(17)	1:06:54:26	(19)

Note, that the runtimes are shown in the format days:hours:minutes:seconds and are followed by the average number of performed iterations.
[§] For rgb2ycbcr, the integer error bounds for 0.25% and 0.50% are equal. Thus, we started from 0.50%.
[†] These benchmarks exceeded the time limit.

MCTS needs to compute more iterations. This will allow MCTS to explore more branches of the search tree to find better solutions. With higher budgets, MCTS will also be able to trade off between exploitation and exploration (cmp. Section 5.3.3). For example, setting the exploration constant (C) to a higher value will cause MCTS to explore branches with low visit counts more often, whereas setting C to a low value will result in MCTS exploiting nodes which proved to be more rewarding in the previous iterations.

Overall, our experiments show that the quality of the result highly depends on the approximation technique as well as on the employed search method. This underlines the necessity of conducting extensive experiments with different approximation and search techniques, which is well supported by CIRCA. Furthermore, our results also point to the fact that the achievable area savings strongly depend on the input design. Some benchmarks, e.g., butterfly or fir_pipe_16, describe challenging approximation problems for which all search methods could achieve only very small area savings.

In Table 4, we list the average runtimes and the average number of performed iterations of the different experiments. For instance, the entry for hc_0_best/aig_wc_25 of the benchmark butterfly under the error bound of 0.25% elaborates that the particular experiment took 1 h, 2 min and 21 s on average and performed a total of four iterations. The formal verification employed in the quality assurance block has been identified as the dominating part in the runtime of the approximation flow. The runtimes of the approximation processes range from a few seconds up to several days, depending on the number of verifications performed and depending on the complexity of the occurring verification problems. However, due to randomness in the taken path through the search space as well as in the applied approximations, the complexity of the occurring verification problems may differ. Thus, the runtime of an approximation process may vary even though the same number of verifications has been performed for the same benchmark circuit, e.g., the butterfly benchmark (hc_0_best/aig_wc_25).

Comparing the runtimes of the three search methods reveals that HC

often provides significantly shorter runtimes than SA or MCTS. However, as Figure 4 shows, the area savings were lower on average as well. Furthermore, HC performs, in general, significantly less iterations compared to SA and MCTS. There are two explanations to this: The first relates to the way we count iterations. Basically, we increase the iteration count once a node gets selected. Compared to HC, SA tends to accept more nodes which naturally increases its iteration count. MCTS performs an iteration when performing back propagation. This leads to more iterations, although the runtime is not increased significantly. Second, HC might get stuck in a local minimum quickly. While HC then terminates the search, SA and MCTS continue with more iterations.

Some of the experiments with HC at an effort level of 6 have exceeded our computation time limit of 14 days. These experiments are marked in Table 4, but are not shown in Figure 4. The excessive runtimes indicate a search space explosion and highlight the importance of choosing an appropriate search method.

7. Conclusion and future work

In this paper, we have presented the CIRCA framework for approximate circuit generation. CIRCA is developed to be a modular and extensible framework, and is publicly available. We have elaborated on the architecture of CIRCA, which was driven by analyzing the commonalities and differences of related frameworks. In our experiments, we have shown CIRCA's ability to employ and exchange functionality, highlighting the advantages of CIRCA's flexible design. We believe that the flexibility of CIRCA will foster comparative studies in the field of Approximate Computing, improving evaluation of different methods and techniques.

We plan future work along several lines: First, we will continue with the implementation of alternative techniques in the QUAES stage, i.e., for quality assurance, approximation, estimation, and search. Besides covering also delay and energy as target metrics, we are particularly interested in the trade-offs between more accurate estimations and the required computational effort. Second, we will focus on the input and output stage. In the input stage, we will look into approaches to automatically identify subcircuits amenable to approximation in the input specification. This would relieve the user from the tedious process of marking potential candidates and make such a framework wider applicable. In the output stage, we plan to connect to back-end synthesis tools, such as FPGA vendor tools or the Synopsys Design Compiler to be able to get accurate circuit characteristics and actually run the approximate circuits, at least in FPGA technology. Third, we want to evaluate how the level of abstraction of a circuit affects the approximation process. Fourth, we want to integrate CIRCA into an high-level synthesis process to automatically generate approximate circuits from a high-level description. Finally, our goal is to extend our publicly available benchmark suite PaderBench, by more circuits to comprise a standard benchmark suite for the Approximate Computing community.

Acknowledgments

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre “On-The-Fly Computing” (SFB 901). Calculations leading to the results presented here were performed on resources provided by the Paderborn Center for Parallel Computing.

References

[1] S. Mittal, A survey of techniques for approximate computing, ACM Comput. Surv. 48 (4) (2016) 62.
[2] S. Venkataramani, K. Roy, A. Raghunathan, Substitute-and-simplify: a unified design paradigm for approximate and quality configurable circuits, Proceedings of the Conference on Design, Automation & Test in Europe, DATE ’13, IEEE, 2013, pp. 1367–1372.
[3] A. Chandrasekharan, M. Soeken, D. Groesse, R. Drechsler, Approximation-aware rewriting of AIGs for error tolerant applications, Proceedings of the 35th International Conference on Computer-Aided Design, ICCAD ’16, ACM, 2016, pp. 83:1–83:8.
[4] K. Nepal, Y. Li, R.I. Bahar, S. Reda, ABACUS: a technique for automated behavioral synthesis of approximate computing circuits, Proceedings of the Conference on Design, Automation & Test in Europe, DATE ’14, IEEE, 2014, pp. 361:1–361:6.
[5] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, A. Raghunathan, ASLAN: synthesis of approximate sequential circuits, Proceedings of the Conference on Design, Automation & Test in Europe, DATE ’14, IEEE, 2014, pp. 364:1–364:6.
[6] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, A. Raghunathan, SALSA: systematic logic synthesis of approximate circuits, Proceedings of the 49th Annual Design Automation Conference, DAC ’12, ACM, 2012, pp. 796–801.
[7] R. Brayton, A. Mishchenko, ABC: an academic industrial-strength verification tool, in: T. Touili, B. Cook, P. Jackson (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, 2010, pp. 24–40.
[8] G. Liu, Z. Zhang, Statistically certified approximate logic synthesis. Proceedings of the 36th International Conference on Computer-Aided Design, ICCAD ’17, IEEE, 2017, pp. 344–351.
[9] D. Sengupta, F.S. Snigdha, J. Hu, S.S. Sapatnekar, SABER: selection of approximate bits for the design of error tolerant circuits, Proceedings of the 54th Annual Design Automation Conference 2017, DAC ’17, ACM, New York, NY, USA, 2017, pp. 72:1–72:6.
[10] A. Chandrasekharan, M. Soeken, D. Große, R. Drechsler, Precise error determination of approximated components in sequential circuits with model checking, Proceedings of the 53rd Annual Design Automation Conference, DAC ’16, ACM, 2016, pp. 1–6.
[11] C. Wolf, Yosys Open SYnthesis Suite, http://www.clifford.at/yosys/ , Accessed date: 9 April 2019.
[12] M. Shafique, W. Ahmad, R. Hafiz, J. Henkel, A low latency generic accuracy configurable adder, Proceedings of the 52nd Annual Design Automation Conference, DAC ’15, ACM, 2015, pp. 86:1–86:6.
[13] V. Mrazek, R. Hrbacek, Z. Vasicek, L. Sekanina, EvoApproxSb: library of approximate adders and multipliers for circuit design and benchmarking of approximation methods, Proceedings of the Conference on Design, Automation & Test in Europe, DATE ’17, IEEE, 2017, pp. 258–261.
[14] G. van Rossum, B. Warsaw, N. Coghlan, PEP 8 — Style Guide for Python Code, https://www.python.org/dev/peps/pep-0008/ , Accessed date: 9 April 2019.
[15] C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, A survey of Monte Carlo tree search methods, IEEE Trans. Comput. Intell. AI Games 4 (1) (2012) 1–43.
[16] B. Kartal, J. Koenig, S.J. Guy, User-driven narrative variation in large story domains using Monte Carlo tree search, Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems, International Foundation for Autonomous Agents and Multiagent Systems, 2014, pp. 69–76.
[17] M. Awais, H. Ghasemzadeh Mohammadi, M. Platzner, An MCTS-based framework for synthesis of approximate circuits, Proceedings of 26th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC ’18), 2018, pp. 219–224.
[18] L. Kocsis, C. Szepesvári, Bandit based Monte-Carlo planning, European Conference on Machine Learning, Springer, 2006, pp. 282–293.
[19] Z. Vasicek, Relaxed equivalence checking: a new challenge in logic synthesis, Proceedings of 2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits & Systems, DDECS ’17, IEEE, 2017, pp. 1–6.
[20] L. Witschen, T. Wiersema, M. Platzner, Making the Case for Proof-carrying Approximate Circuits, Presented at the 4th Workshop on Approximate Computing (WAPCO ’18) 2018, Manchester, England, 2018, Jan. 2018.
[21] B. Reynwar, Decimation-In-Time fast Fourier Transform, https://github.com/benreynwar/fft-dit-fpga , Accessed date: 9 April 2019.
[22] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K.B. Kent, J. Anderson, J. Rose, V. Betz, VTR 7.0: next generation architecture and CAD system for FPGAs, ACM T. ReconFigure Techn. 7 (2) (July 2014) 6:1–6:30.
[23] Altera Corporation, Altera Advanced Synthesis Cookbook, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/stx_cookbook.pdf , Accessed date: 9 April 2019.
[24] D. Lundgren, OpenCores jpegencode, https://github.com/chiggs/oc_jpegencode , Accessed date: 9 April 2019.