Jump Search: A Fast Technique for the Synthesis of Approximate Circuits

Linus Witschen
Paderborn University, Germany
witschen@mail.upb.de

Hassan Ghasemzadeh Mohammadi
Paderborn University, Germany
hgm@mail.upb.de

Matthias Artmann
Paderborn University, Germany
martmann@mail.uni-paderborn.de

Marco Platzner
Paderborn University, Germany
platzner@mail.upb.de

ABSTRACT

State-of-the-art frameworks for generating approximate circuits automatically explore the search space in an iterative process - often greedily. Synthesis and verification processes are invoked in each iteration to evaluate the found solutions and to guide the search algorithm. As a result, a large number of approximate circuits is subjected to analysis - leading to long runtimes - but only a few approximate circuits might form an acceptable solution.

In this paper, we present our *Jump Search* (JS) method which seeks to reduce the runtime of an approximation process by reducing the number of expensive synthesis and verification steps. To reduce the runtime, JS computes impact factors for each approximation candidate in the circuit to create a selection of approximate circuits without invoking synthesis or verification processes. We denote the selection as *path* from which JS determines the final solution. In our experimental results, JS achieved speed-ups of up to 57x while area savings remain comparable to the reference search method, Simulated Annealing.

CCS CONCEPTS

• **Hardware → Logic synthesis**; **Methodologies for EDA**; **Software tools for EDA**; *Emerging technologies.*

KEYWORDS

Approximate computing; design automation; parameter selection; circuit synthesis

ACM Reference Format:
Linus Witschen, Hassan Ghasemzadeh Mohammadi, Matthias Artmann, and Marco Platzner. 2019. Jump Search: A Fast Technique for the Synthesis of Approximate Circuits. In *Great Lakes Symposium on VLSI 2019 (GLSVLSI '19), May 9–11, 2019, Tysons Corner, VA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3299874.3317998

GLSVLSI '19, May 9–11, 2019, Tysons Corner, VA, USA

ACM ISBN 978-1-4503-6252-8/19/05…$15.00
https://doi.org/10.1145/3299874.3317998

1 INTRODUCTION

Many applications in various domains either exhibit an inherent error resilience, e.g., multimedia processing, or do not provide a golden result, e.g., recommender systems. The design paradigm Approximate Computing utilizes this fact to trade off a circuit's output quality against a target metric, e.g., hardware area, delay, or energy consumption. Approximations can be applied at all levels of the computing stack, from the software level down to the logic and circuit level [7]. In this work, we are focusing on the circuit level and, more precisely, on the automatic as well as efficient generation of *Approximate Circuits* (AxCs).

Frameworks to automatize the process of generating AxCs have been presented in the past [6, 8, 9, 12, 13]. The approximation process states an optimization problem with various objectives such as hardware area, delay, or energy consumption. A major issue for the optimization problem is that only very few general assumptions can be made about the search space. Thus, to find an optimal solution (i.e., an optimal AxC), iterative heuristic-based search methods are employed and the search space is expanded step-by-step - often greedily. The solutions (i.e., AxCs) found are then evaluated and ranked. For the evaluation of the AxCs, current frameworks employ synthesis and verification steps. Since the search space is usually large, a large number of solutions is found and thus a large number of evaluation steps is invoked. However, the evaluation step generally shows a considerable runtime - especially the verification step. In fact, the evaluation dictates the runtime of the framework and, in general, it applies that the more synthesis and verification steps are invoked, the longer is the runtime of the approximation process. Many of the AxCs found, however, are not of interest, i.e., are beyond the acceptable quality.

In this paper, we seek to minimize the runtime of the approximation process with our novel approach *Jump Search* (JS). JS executes two phases to achieve this goal:

(1) Pre-select AxCs from the search space by using a heuristic function without invoking synthesis or verification steps.
(2) For evaluation, invoke synthesis and verification steps only for judiciously chosen AxCs to minimize the runtime.

In the first phase, JS utilizes a heuristic function to select the most promising AxCs from all AxCs in the search space. The selected AxCs are sorted by their depth in the search space, where a deep depth results in more approximations applied to the circuit. For each depth, JS selects only one AxC. We denote the set of selected AxCs as *path*, since the set represents a sequence of AxCs, from the original circuit towards the boundaries of the search space,

i.e., towards AxCs with more approximations applied. The runtime of the first phase is short since no synthesis or verification steps are invoked. In the second phase, JS searches on the path for the AxC with the deepest depth, i.e., the AxC with the most aggressive approximations applied, which satisfies the user-defined quality constraints. The search in the second phase of JS is allowed to *jump* over AxCs to find the resulting AxC more efficiently. Since expensive synthesis and verification steps are only invoked in the second phase, JS is able to significantly reduce the runtime. A disadvantage of this approach is that the search might jump over an optimal solution. However, we anticipate two benefits from our approach: 1) we can quickly find an acceptable AxC and evaluate the potential of savings in a circuit; and 2) we find a *good* starting point in the search space for a subsequent, more thoroughly operating search method, fine-tuning the AxC found by JS.

The path selected in the first phase of JS is affecting the quality of the final outcome. Thus, the employed heuristic function is key. In our novel approach, JS uses parameter selection techniques to determine the impact of each *candidate* - a subcircuit subject to approximations - on the quality of the overall circuit. Furthermore, an impact factor for the area is determined for each candidate. Our experimental results show that JS can achieve speed-ups of up to 57x while area savings remain comparable to the reference algorithm.

The remainder of this paper is structured as follows. In Section 2, we discuss related work. In Section 3, we present the details of our parameter selection and of our JS approach. The effectiveness of JS is highlighted in our experimental results in Section 4. Finally, we conclude the paper and describe the future work in Section 5.

2 RELATED WORK

Various frameworks for the automated generation of AxCs have been presented in the past. Venkataramani et al. [13] presented SALSA that forms a so-called *quality constraint circuit*, comprising of the original circuit, the AxC, and the quality function. The primary inputs of the quality constraint circuit are fed into the original circuit and the AxC. The respective output of the original circuit and the AxC form the inputs of the quality function. The quality function has a single output bit, indicating whether the AxC violates the quality constraints. Initially, the AxC is identical to the original circuit. Rather than employing a search technique, SALSA first identifies the *approximate don't cares* of the quality function. Then, standard don't care optimization techniques are used to approximate the circuit while ensuring that the quality constraints are satisfied. In this way, SALSA generates combinational AxCs which adhere to the quality constraints by construction.

Venkataramani et al. also presented SASIMI [12], a framework which uses *substitute-and-simplify* as approximation technique. SASIMI identifies near-identical signals, i.e., signals that show a similar behavior, and substitutes one with the other to simplify the logic. The identified signals are ranked using a heuristic function which considers area and delay parameters. Hill climbing search selects the highest ranked signal pair for substitution until the quality constraints are violated and the search terminates. In each iteration of the search, SASIMI carries out synthesis and simulation processes to identify signal pairs and to evaluate the AxC's error.

Chandrasekharan et al. [4] proposed to transform the logic function of the combinational input circuit into an *AND-Inverter Graph* (AIG) representation and to identify the critical path in the AIG. Then, a search algorithm greedily selects the node on the critical path with the smallest cut size and replaces it with constant 0. After each replacement, a SAT solver formally verifies the quality of the AxC. The process terminates if the AxC violates the quality constraints or the maximum number of iterations is reached.

Nepal et al. [8] proposed the ABACUS methodology which operates on the behavioral description of the input circuit and transforms the circuit into an *Abstract Synthesis Tree* (AST). Starting from the original circuit, ABACUS iteratively applies transformations to the currently selected AST to create different AxCs. In each iteration of ABACUS, the AxCs are synthesized to determine area and power characteristics, and testing evaluates the accuracy of each circuit. The resulting three circuit parameters are combined into a fitness function. The circuit with the best fitness is greedily selected as the next current circuit. The process terminates after a user-defined number of iterations, and the Pareto frontier, trading off accuracy for power, is provided to the user.

The ASLAN framework by Ranjan et al. [9] approximates sequential input circuits and provides a guarantee that the resulting AxC never violates the quality constraints. In a pre-processing step, ASLAN identifies subcircuits in the input circuit amenable to approximations, so-called *candidates*. Then, different approximated variants of each candidate are generated by applying varying quality constraints to each candidate and different approximations. The resulting variants differ in their quality and energy consumption; the ensemble of all variants form ASLAN's search space. Finally, ASLAN performs hill climbing search to find the combination of approximated candidates that minimizes the energy consumption of the overall circuit. ASLAN pre-generates the entire search space, traverses it iteratively, and invokes verification in each iteration.

Liu and Zhang's SCALS framework [6] initially maps the input gate-level logic to a target technology. In an iterative process, various AxCs are generated by extracting subnetlists from the mapped netlist which are then subjected to randomly chosen approximations or optimizations. The AxCs generated in each iteration are then evaluated based on their error, determined via testing, and their area. A Metropolis-Hastings algorithm controls the selection of AxCs and terminates after a defined number of iterations.

Soeken et al. [10] represent the circuit description as a BDD and apply approximations to reduce the size of the BDD. Froehlich et al. [5] also utilize a BDD representation to determine an optimal BDD, i.e., a BDD with a minimum number of nodes, satisfying the quality constraints. Using BDDs allows the authors to make very precise statements about the resulting error. The major weakness of these approaches, however, is that BDDs are only applicable to small, combinational circuits. Thus, the approaches may be applied to small subcomponents of a design, which would then require a subsequent quality verification process for the overall design.

Barbareschi et al. [2] proposed $\mathbb{IDEA}$ as a high-level synthesis framework in which a branch-and-bound-based algorithm selects the appropriate approximation for the candidates from all possible ones. In each iteration, $\mathbb{IDEA}$ utilizes a depth-first search strategy and examines all approximation possibilities for a target candidate until no more possible approximations are available. Afterwards,

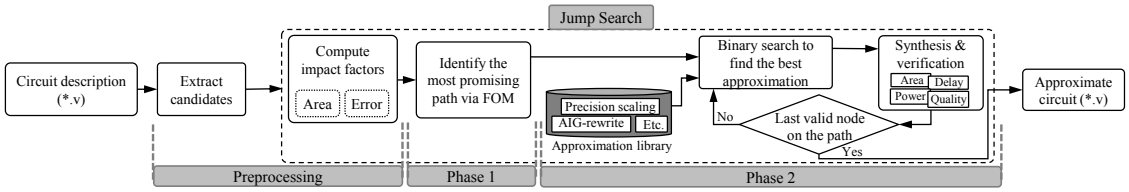

Figure 1: Overview of the Jump Search synthesis framework

IDEA backtracks. IDEA spends the largest amount of the search budget on identifying the best possible approximation of the early selected candidates in the design.

Awais et al. [1] presented a framework which utilizes a *Monte-Carlo Tree Search* (MCTS) algorithm in order to identify the target candidates for approximation as well as their appropriate approximations, given a specified search budget. MCTS selects a candidate and iteratively applies different types of approximations to it until the algorithm identifies other promising candidates in the design space through a number of heuristic functions. The new candidates are then iteratively subjected to different approximations. The process continues until the given search budget is exhausted. The employed heuristic functions limit the applicability of the framework, since the functions rely on several hyper-parameters, which have to be tuned for each circuit before the synthesis process.

The approximation techniques or frameworks presented in the past heavily rely on synthesis and verification results, either to guide the search, e.g., ASLAN, or to evaluate randomly generated AxCs, e.g., SCALS, leading to considerable runtimes.

Our JS method seeks to reduce the runtime of the approximation process with two novel approaches:

(1) Prior to the approximation process, JS determines the candidates' impacts on the overall output quality using parameter selection techniques and the candidates' impacts on the overall circuit area;
(2) JS pre-selects a set of AxCs based on a heuristic evaluation, and thus, avoids excessive invocations of synthesis and verification processes.

3 THE SYNTHESIS OF APPROXIMATE CIRCUITS USING JUMP SEARCH

In this section, we explain the concepts of JS, a fast synthesis method for AxCs. We first describe the workflow of the proposed synthesis framework. Then, we explain the details of impact factors and JS.

3.1 Overview

The goal of JS is to reduce the runtime of an approximation process while achieving area savings comparable to other, more exhaustive search techniques. JS achieves reductions in runtime mainly through the minimization in the number of synthesis and verification steps, as these are usually the most expensive tasks.

The proposed framework is illustrated in Figure 1. In the preprocessing step, JS identifies subcircuits in the input circuit amenable to approximations, usually arithmetic components. The subcircuits are denoted as *candidates*. In the next step, the candidates are analyzed in terms of their impact on the area and the precision of the design. The goal of the analysis is to start approximations from the candidates that have the highest impact on the circuit size and the lowest impact on the quality of the design. In fact, the costly search-based optimization process is strictly controlled to exclude unpromising candidates. For each candidate, two impact factors are determined: 1) the candidate's impact on the overall area of the circuit (denoted as if_{area}) and 2) the candidate's impact on the overall output quality or error, respectively, of the circuit (denoted as if_{err}).

To determine a candidate's if_{area}, the particular candidate's output is tied to constant 0 and the overall circuit is synthesized. The relative saving in area represents the candidate's if_{area} and ranges from 0 (the candidate has no impact on the circuit area) to 1 (the candidate renders the overall circuit logic obsolete).

A candidate's if_{err} is determined with a parameter ranking method described in more detail in the next section. Here, the basic idea is to sample the outputs of each candidate as well as of the primary output(s) of the input design. From the sampled output values a dataset is compiled, containing input and output parameters. The input parameters are represented by the outputs of the candidates. The output parameters are represented by the primary output(s) of the input design. In a testbench, uniformly distributed random values are assigned to the primary inputs of the input circuit to generate different input and output parameters. The sampled parameters are then compiled into the dataset which is processed by the ranking method to provide JS with the contribution of each candidate on the quality of the primary output(s).

Throughout the approximation process, various AxCs are produced, comprising of different approximated versions of the candidates. The impact factors guide JS towards the selection of AxCs which apply aggressive approximations on the most promising candidates, i.e., those candidates which have a high if_{area} and a low if_{err}. From the selected AxCs, a modified binary search finds the AxC that satisfies the user-defined quality constraints and is subjected to the most aggressive approximations, and thus, potentially large savings in the target metric. To find such an AxC, synthesis and verification steps are invoked for a precise evaluation of the AxC. Due to the pre-selection of AxCs of interest, which only relies on pre-computed parameters, and the binary search, the number of expensive synthesis and verification steps is reduced in JS. Finally, the Verilog code of the most promising AxC is generated.

3.2 The Parameter Selection

The impact of a candidate on the error (i.e., if_{err}) is estimated via a parameter selection operator. For this purpose, *Least Absolute Shrinkage and Selection Operator* (LASSO) [11] is exploited to rank the candidates according to their contributions on the circuit

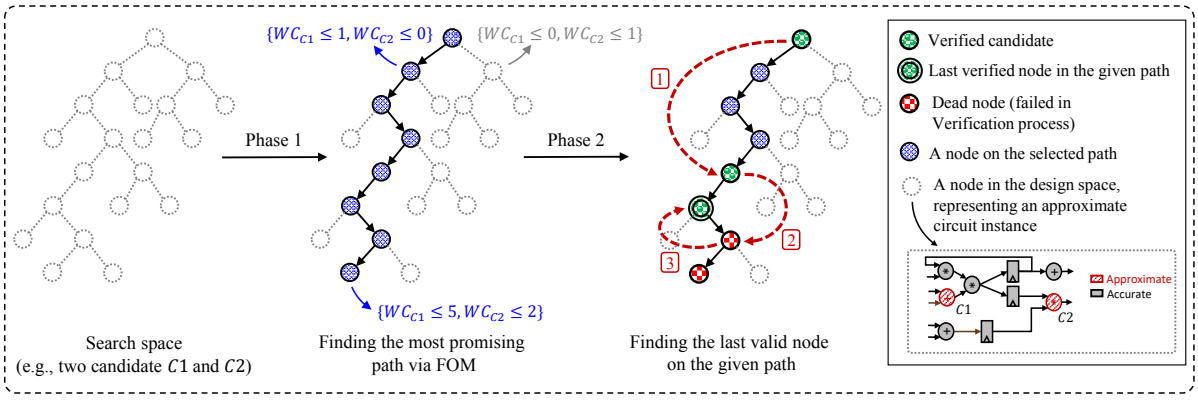

Figure 2: Visualization of the Jump Search concept.

output(s). LASSO, as a linear regression technique, uses l_1−norm regularization in its objective function and ranks the parameters subject to the following objective function:

$$min \sum_{i=1}^{n} \{\frac{1}{2} \left\| y_i - \mathbf{w}^T \mathbf{x}_i \right\|_2^2 + \lambda \left\| \mathbf{w} \right\|_1\} \quad (1)$$

where $y_i \in \mathbb{R}$ is the output sample of the circuit, $\mathbf{w} \in \mathbb{R}^n$ is the vector of regression coefficients, and $\mathbf{x}_i \in \mathbb{R}^n$ refers to a vector of input parameters. Moreover, $\lambda \in \mathbb{R}$ denotes the regularization hyper-parameter and controls the values of the regression coefficients. In case of a multiple-output circuit, we use an aggregation function, e.g., weighted arithmetic mean, to estimate the contribution of each input parameter. For multiple-fanout candidates, we take into account each output as a separate input parameter. Here, the maximum value of the corresponding coefficients is selected as the impact factor of the candidate.

The hyper-parameter λ controls the values and the number of non-zero coefficients. A large value for λ decreases the number of non-zero elements in $\mathbf{w}$ and hence keeps the value of the objective function small. It should be noted that a small value in Equation 1 does not necessarily mean a small modeling error and therefore a preferred ranking. The linear regression along with the penalty term implies a consistent parameter ranking. Thus, we just tune λ in a way that the number of non-zero coefficients is maximized. The absolute values of the coefficients then represent the contribution of each candidate to the output and are used as if_{err} in JS' heuristic function.

3.3 The Jump Search

Algorithm 1 shows the pseudocode of JS. JS takes the original circuit i together with user-defined quality constraints ϵ as inputs. In a pre-processing step (lines 4-8), the candidates C, usually arithmetic components in the original circuit i, are identified. For each of the identified candidates, if_{area} (line 6) and if_{err} (line 8) are determined with the approaches described previously.

After the pre-processing, the first phase of JS is executed (lines 9-15). In this phase, JS iteratively plans a path P through the search space in which each step on the path represents a different AxC. The candidates define the dimensions of the search space and each candidate is described by a *local error bound*, which comprises of an error metric (we consider the *Worst-Case Error* (WC)) and an error threshold. This approach enables JS to expand the search space in a controlled way and each solution or AxC is uniquely defined.

Figure 2 (left tree) visualizes this at the example of a circuit with two candidates, *C1* and *C2*. The original circuit *i*, for example, is described by $\{WC_{C1} \leq 0,\ WC_{C2} \leq 0\}$. Increasing the local error bounds of the candidates presumably leads to a reduced quality of the overall circuit but also to larger area savings. Starting from the original circuit *i* ($\{WC_{C1} \leq 0,\ WC_{C2} \leq 0\}$), a possible next step on the path is defined by modifying the local error bound of one candidate (lines 9 and 14). For the exemplary circuit in Figure 2 (center tree), $\{WC_{C1} \leq 1,\ WC_{C2} \leq 0\}$ and $\{WC_{C1} \leq 0,\ WC_{C2} \leq 1\}$ are the possible next steps; blue circles highlight the planned path. JS plans the path by selecting the AxC with the highest *Figure of Merit* (FOM) (line 12). The FOM is computed by the heuristic function in Equation 2 (line 11).

```
   Input: i= description of the circuit, ϵ= quality constraints
   Output: a= approximate instance satisfying quality constraints
1  Function jumpSearch(i, ϵ):
2      C ← ∅; // Set of candidates
3      P ← { i }; // Path, each step represents an AxC
       /* Pre-processing                                          */
4      C ← identifyCandidates( i );
5      foreach c ∈ C do
6          c.if_area ← impactArea( c );
7      end
8      computeImpactError( i, C );
       /* Phase 1: path planning                                  */
9      next_steps ← getNextSteps( i )
10     while next_steps do
11         evaluateSteps( next_steps ); // Compute FOM of each step
12         n ← getBestStep( next_steps );
13         P.append( n );
14         next_steps ← getNextSteps( n );
15     end
       /* Phase 2: binary search                                  */
16     l ← 0; u ← len(P)-1;
17     while l ≠ u do
18         m= (u+l)/2 + 1; a ← P[m];
19         synthesizeAndVerify( i, a, ϵ );
20         if isValid( a ) then l ← m ;
21         else u ← m-1; a ← P[l] ;
22     end
23     return a;
```

Algorithm 1: Pseudocode of the Jump Search algorithm.

$$FOM(S) = \sum_{c} \sqrt{\frac{Err(c)}{MaxE(c)}} \times \frac{IF_{Area}(c)}{|IF_{Err}(c)|+1} \quad (2)$$

The heuristic estimates the FOM of each step S for each given candidate c. For each candidate, the applied error threshold *Err*, the maximal possible error of the candidate *MaxErr*, as well as the if_{area} and if_{err} are taken into account. Note that each parameter is a pre-computed value; thus, the first phase does not invoke any synthesis or verification processes. The first phase plans a path that increases the depth, i.e., the level of approximation, with each step. The phase terminates when no more steps are available, i.e., the end of the search space is reached.

The heuristic function guides JS towards AxCs which reduce the quality of the candidates with a low impact on the overall circuit quality but a high impact on the circuit area. The term under the square root balances the quality reduction of the candidates to avoid only reducing the quality of certain candidates.

In the second phase, JS performs a binary search on path P to find the AxC with the deepest depth which still satisfies the user-defined quality constraints ϵ (line 16-22). JS seeks to find the deepest AxC since we assume that this AxC was subjected to the most aggressive approximations, and thus, shows high area savings - which might not be optimal but significant. Figure 2 (right tree) visualizes the second phase. Path P from the first phase is highlighted in blue, the valid AxCs are highlighted in green and invalid AxCs in red. The binary search maintains three indicies: a lower bound l, an upper bound u, and the middle index m. The search synthesizes and verifies the AxC at step m on the path P (line 18-19) and adjusts the lower bound l and upper bound u accordingly (line 20 and 21). In Figure 2 (right tree), the red arrows point to the middle index of the search, i.e., the AxC subjected to evaluation; the circled red numbers indicate the execution order. The search terminates when l and u are equal, i.e., the deepest AxC is found which satisfies the quality constraints (highlighted by the green circle with doubled outlines). In the second phase, JS invokes synthesis and verification process as this is inevitable to evaluate the AxCs (line 19-21). However, compared to related work, JS significantly reduces the number of invoked synthesis and verification process due to the pre-selection of AxCs in the first phase.

4 EXPERIMENTAL SETUP AND RESULTS

For the evaluation of our approach, we have used a subset of the approximate computing benchmark set, PaderBench[1], comprising of six circuits. Firstly, we simulated each circuit with a dataset of 10,000 random vectors and sampled the output values of the candidates as well as the circuit's output. From the sampled data, LASSO computes the if_{err} of each candidate. Secondly, for each candidate, the output is tied to constant 0 and ABC [3] is invoked to report on the circuit's FPGA 4-LUT usage to determine the candidate's if_{area}.

The JS algorithm has been implemented and is available in our approximation framework, CIRCA[2] [14]. For each benchmark circuit, we have applied the worst-case error metric, varying the error bound from 0.50% to 5.00% of the maximal possible output value,

[1]https://go.upb.de/paderbench
[2]https://go.upb.de/circa

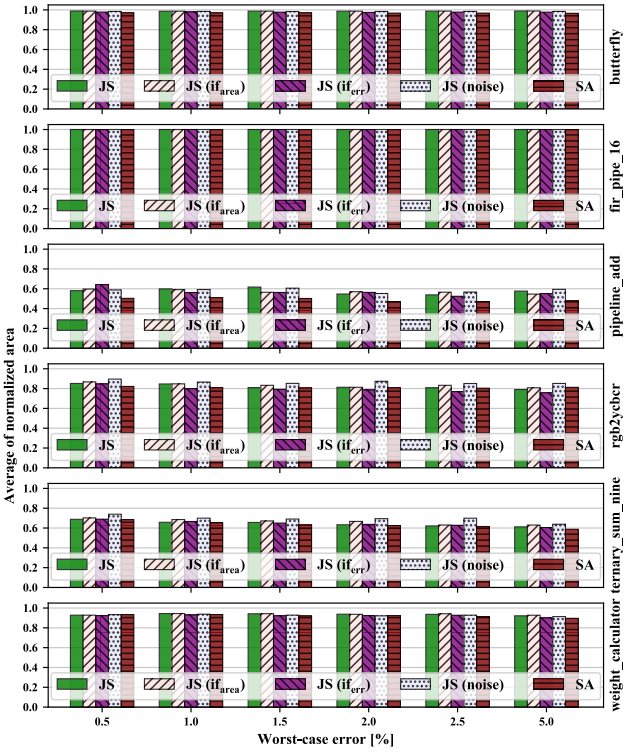

Figure 3: Normalized area results.

and performed four experiments to evaluate the influence of different impact factors on the results: JS performs (1) with the pre-computed impact factors, if_{area} and if_{err}; (2) with noise added to the impact factors to emulate a random path; (3) with if_{area} as the only impact factor taken into account (if_{area} in the results); and (4) with if_{err} as the only impact factor taken into account (if_{err} in the results). *Simulated Annealing* (SA) has been used as a reference implementation, since previously performed experiments have shown that the algorithm achieves significant area savings.

We have run each experiment ten times and report the averaged normalized area as well as runtimes. The experiments have been performed on a compute cluster which runs Scientific Linux 7.2 (Nitrogen), provides 2 Gigabyte of main memory per job, and comprises nodes with an Intel® Xeon E5-2670@2.6GHz (16 cores).

Figure 3 shows the experimental results. Both JS and SA achieve area savings, and, for all benchmarks, JS achieves savings comparable to SA. *pipeline_add*, however, states an exception where the area savings of JS are significant, yet slightly worse than SA's. Furthermore, the results show that taking a random path already leads to notable area savings on average; taking into account the provided impact factors, however, leads to better results on average. Comparing the results of if_{area} and if_{err} shows that following only the guidance of if_{err} leads to slightly better results in the majority of the cases (sometimes even better than SA), suggesting that the impact of if_{err} on the path planning could be increased by a weight.

Table 1 shows the average runtimes of the experiments. The table reveals that JS achieves speed-ups of up to 57x and clearly outperforms SA in this regard. Overall, while the area savings on

average remain comparable, JS' runtime is significantly shorter due to the reduction of synthesis and verification steps. Furthermore, considering the runtime as a budget parameter of the approximation process, JS utilizes only a fraction of SA's budget requirement. Hence, to gain further area savings, we imagine a third phase in JS which utilizes the remaining budget to fine tune the AxC from the second phase. In other words, the first and second phase of JS become pre-processing steps to quickly find a good starting point for the actual search, e.g., SA, which further alters the AxC, but more thoroughly.

Table 1: Average runtimes of the search approaches.

Search method	Worst-case error [%] 0.50	1.0	1.50	2.00	2.50	5.00
butterfly						
JS	17:20	16:52	17:11	17:16	17:17	17:07
JS (if_{area})	17:04	16:15	16:46	17:22	16:46	17:19
JS (if_{err})	15:50	16:59	17:12	16:48	16:23	16:44
JS (noise)	17:22	17:26	17:30	17:00	17:01	16:43
SA	3:53:18	3:46:56	3:46:25	3:48:41	3:52:19	3:57:30
fir_pipe_16						
JS	13:53	13:25	13:46	13:27	13:25	13:33
JS (if_{area})	13:19	13:23	13:34	13:10	12:56	14:01
JS (if_{err})	14:08	13:18	13:31	14:15	13:52	13:37
JS (noise)	13:06	13:10	12:35	13:12	12:24	13:33
SA	3:40:32	3:43:39	3:27:02	3:43:14	3:34:36	3:16:00
pipeline_add						
JS	29	25	25	27	34	24
JS (if_{area})	25	23	26	25	24	27
JS (if_{err})	25	26	23	24	23	23
JS (noise)	28	25	26	24	23	35
SA	2:31	2:32	2:36	2:45	2:44	2:46
rgb2ycbcr						
JS	7:11	7:07	7:31	7:47	7:40	8:04
JS (if_{area})	7:13	7:23	7:31	7:46	7:31	7:40
JS (if_{err})	7:22	7:42	8:09	8:09	8:05	8:06
JS (noise)	7:29	7:47	8:01	8:00	7:47	9:18
SA	4:27:34	4:47:00	4:46:55	5:58:23	5:36:35	7:16:34
ternary_sum_nine						
JS	44	41	43	44	41	45
JS (if_{area})	40	43	44	38	39	43
JS (if_{err})	41	39	40	41	39	39
JS (noise)	47	41	40	41	44	43
SA	4:00	4:23	5:54	5:12	5:07	5:02
weight_calculator						
JS	14:37:26	11:53:40	4:37:38	7:18:44	11:18:25	8:30:48
JS (if_{area})	14:38:33	13:12:27	6:37:08	8:03:40	13:00:33	9:20:41
JS (if_{err})	14:41:01	12:44:24	13:49:32	10:02:02	14:26:41	9:02:23
JS (noise)	11:14:21	13:47:01	6:59:13	8:46:52	4:24:13	6:47:21
SA	1:04:36:58	1:08:16:42	1:10:05:39	1:07:29:23	1:10:43:29	1:01:34:45

The runtimes are shown in the format days:hours:minutes:seconds.

5 CONCLUSION AND FUTURE WORK

In this work, we have presented our Jump Search method - an approach to reduce the runtime of approximation processes. JS utilizes parameter selection techniques to pre-compute impact factors for the approximation candidates of a circuit. The first phase of JS uses the impact factors to plan a path through the search space without invoking expensive synthesis or verification steps. The path, i.e., the resulting set of AxCs of interest, is evaluated in the second phase to determine the outcome of JS. In our experimental results, we have achieved significant reductions in the runtimes of the approximation processes while providing area savings comparable to a reference method, Simulated Annealing. Even though JS' area savings are slightly worse on average compared to SA, JS is beneficial for two reasons: 1) JS returns an AxC which improves the target metric, enabling the swift evaluation of potential for savings in a circuit and 2) JS can be employed as a pre-processing step to find a good starting point in the search space from where the found AxC can be fine tuned. JS achieves the reduction in runtime mainly because of its novel approach to utilize parameter selection techniques to pre-compute impact factors which allow for the evaluation of AxCs without invoking synthesis or verification.

In the future, we want to compare different parameter selection techniques and extend JS by a third phase, which uses the second phase's endpoint as a starting point, and fine tunes the found AxC. Finally, we envision JS to introduce checkpoints in the first phase. At such checkpoints, JS can update its heuristic and, if necessary, re-plan the path, i.e., backtrack in the search space.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the funding of this project by computing time provided by the Paderborn Center for Parallel Computing (PC2).

REFERENCES

[1] Muhammad Awais, Hassan Ghasemzadeh Mohammadi, and Marco Platzner. 2018. An MCTS-based Framework for Synthesis of Approximate Circuits. In *Proceedings of the Conference on Very Large Scale Integration (VLSI-SoC '18)*. IEEE, 219–224.

[2] Mario Barbareschi, Federico Iannucci, and Antonino Mazzeo. 2016. Automatic design space exploration of approximate algorithms for big data applications. In *Proceedings of the 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA '16)*. IEEE, 40–45.

[3] Robert Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer Berlin Heidelberg, 24–40.

[4] Arun Chandrasekharan, Mathias Soeken, Daniel Groesse, and Rolf Drechsler. 2016. Approximation-aware Rewriting of AIGs for Error Tolerant Applications. In *Proceedings of the 35th International Conference on Computer-Aided Design (ICCAD '16)*. ACM, Article 83, 8 pages.

[5] Saman Froehlich, Daniel GroBe, and Rolf Drechsler. 2017. Error Bounded Exact BDD Minimization in Approximate Computing. In *Proceedings of the 47th International Symposium on Multiple-Valued Logic (ISMVL '17)*. IEEE, 254–259.

[6] Gai Liu and Zhiru Zhang. 2017. Statistically certified approximate logic synthesis. In *Proceedings of the 36th International Conference on Computer-Aided Design (ICCAD '17)*. IEEE, 344–351.

[7] Sparsh Mittal. 2016. A Survey of Techniques for Approximate Computing. *ACM Computing Surveys (CSUR)* 48, 4, Article 62 (March 2016), 33 pages.

[8] Kumud Nepal, Yueting Li, R. Iris Bahar, and Sherief Reda. 2014. ABACUS: A Technique for Automated Behavioral Synthesis of Approximate Computing Circuits. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE '14)*. IEEE, Article 361, 6 pages.

[9] Ashish Ranjan, Arnab Raha, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. 2014. ASLAN: Synthesis of Approximate Sequential Circuits. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE '14)*. IEEE, Article 364, 6 pages.

[10] Mathias Soeken, Daniel Große, Arun Chandrasekharan, and Rolf Drechsler. 2016. BDD minimization for approximate computing. In *Proceedings of the 21st Asia and South Pacific Design Automation Conference (ASP-DAC '16)*. IEEE, 474–479.

[11] Robert Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)* (1996), 267–288.

[12] Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. 2013. Substitute-and-simplify: A Unified Design Paradigm for Approximate and Quality Configurable Circuits. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE '13)*. IEEE, 1367–1372.

[13] Swagath Venkataramani, Amit Sabne, Vivek Kozhikkottu, Kaushik Roy, and Anand Raghunathan. 2012. SALSA: Systematic Logic Synthesis of Approximate Circuits. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*. ACM, 796–801.

[14] Linus Witschen, Tobias Wiersema, Hassan Ghasemzadeh Mohammadi, Muhammad Awais, and Marco Platzner. 2018. CIRCA: Towards a Modular and Extensible Framework for Approximate Circuit Generation. (2018), 6 pages. Presented at the 3rd Workshop on Approximate Computing (AxC '18).