SCGen: A Versatile Generator Framework for Agile Design of Stochastic Circuits

Zexi Li[1], Haoran Jin[1], Kuncai Zhong[2], Guojie Luo[3,4], Runsheng Wang[5], and Weikang Qian[1,6]
[1]University of Michigan-SJTU Joint Institute and [6]MoE Key Lab of AI, Shanghai Jiao Tong University, China
[2]College of Semiconductors, Hunan University, China
[3]School of Computer Sciences and [4]Center for Energy-Efficient Computing and Applications, Peking University, China
[5]School of Integrated Circuits, Peking University, China
Emails: {lzx12138, allenjin}@sjtu.edu.cn, kczhong@hnu.edu.cn, {gluo, r.wang}@pku.edu.cn, qianwk@sjtu.edu.cn

***Abstract*—Stochastic computing (SC) is an unconventional computing paradigm with unique features. Designing SC circuits is dramatically different from designing binary computing (BC) circuits. To support the agile design of SC circuits, we propose SCGen, a versatile generator framework, which provides users with a C++ interface to easily specify SC circuits and supports 1) accelerated accuracy simulation, 2) accelerated design space exploration (DSE) for accuracy maximization guided by simulated annealing (SA) and genetic algorithm (GA), 3) circuit optimization by random number source (RNS) sharing, 4) circuit verification via symbolic expression analysis, and 5) automatic Verilog code generation. Furthermore, we extend SCGen to also support agile design of hybrid SC-BC circuits. The experimental results show that our proposed DSE acceleration methods achieve up to 59× speedup, the DSE with SA and GA can get an average reduction of 4.0% and 12.7%, respectively, in accuracy loss compared to random search, and RNS sharing reduces the average area and power by 41% and 47%, respectively.**

***Index Terms*—stochastic computing, agile design, accuracy simulation, design space exploration, simulation acceleration**

I. Introduction

Stochastic computing (SC) has gained increasing attention as a novel computing paradigm these days [1]. Different from conventional binary computing (BC) where numbers are encoded in binary radix form, SC represents numbers as the ratio of 1s in a stochastic bit stream [2]. For example, a 4-bit stream "1010" encodes the value 0.5. Compared to BC, SC has the advantage in fault tolerance, as all bits have equal weights, and therefore, the number does not vary much if some bits are flipped accidentally [1]. Moreover, computing units in SC are simpler. For example, the multiplication in SC is implemented using a single AND gate as shown in Fig. 1a. With these advantages, SC has been applied in a few domains such as image processing [3] and neural network [4].

Compared to BC, SC has many unique features. For example, SC needs a special module called stochastic number generator (SNG) to produce a stochastic bit stream. Moreover, unlike BC, which always produces accurate results, SC is subject to some inaccuracy in its output. Fig. 1b shows a case where the computing result is incorrect. Thus, in the current design practice of SC circuits, besides normal hardware cost metrics such as area and delay, accuracy is also considered.

Due to these unique features, designing an SC circuit is dramatically different from designing a BC circuit. To facilitate its design, it is desired to have a design tool that can automatically produce the register-transfer level (RTL) description of an SC circuit from a higher-level specification. For this purpose, Alaghi designed *scsynth* that can convert an arithmetic function specified in Matlab into an SC circuit described in RTL [5]. However, it does not support accuracy simulation for SC circuits. To address this issue, Daruwalla *et al.* proposed *BitSAD v2*, which supports both accuracy simulation and Verilog code generation [6]. However, it still lacks verification of the correctness of the generated SC circuit. Besides, as an SC circuit has many tunable knobs that can lead to different accuracies, it is desired to perform design space exploration (DSE) over these knobs to maximize the accuracy, but BitSAD v2 does not support DSE for accuracy maximization. Finally, it does not support the design of *hybrid SC-BC circuits*, which contain both SC and BC modules and are used in image processing and neural networks, such as Gaussian blur (GB) [7] and convolution circuits [8].

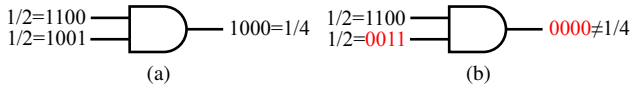

Fig. 1. SC multiplication using a single AND gate: (a) correct result; (b) incorrect result.

To address the above issues, in this work, we propose SCGen, a versatile generator framework for agile design of SC circuits. Our contributions are as follows:

- We implement a basic framework that provides users with a C++ interface to easily specify SC circuits and supports accuracy simulation, DSE for accuracy maximization, circuit optimization by random number source (RNS) sharing, circuit verification, and automatic Verilog code generation.
- We propose several methods to speed up the accuracy simulation and DSE for SC circuits, and two effective DSE methods for accuracy maximization based on simulated annealing (SA) and genetic algorithm (GA), respectively.
- We extend SCGen to support hybrid SC-BC circuit design. Particularly, we propose an RNS sharing method for hybrid SC-BC circuits to reduce area and power.

The experiment results show that our proposed acceleration methods can achieve up to 59× speedup. The DSE with SA and GA can get an average reduction of 4.0% and 12.7%, respectively, in accuracy loss compared to random search. The RNS sharing can reduce the average area and power by 41% and 47%, respectively. The code of SCGen is made open-source at https://github.com/SJTU-ECTL/SCGen.

This work is supported by the National Key R&D Program of China under grant number 2020YFB2205501. Zexi Li and Haoran Jin contributed equally. Corresponding authors: Runsheng Wang and Weikang Qian.

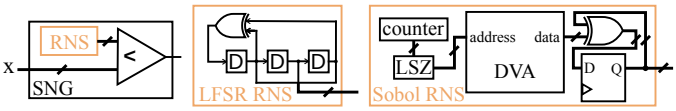

Fig. 2. Structure of the SNG with a 3-bit LFSR RNS or Sobol RNS, where LSZ stands for *least significant zero detector* and DVA stands for *direction vector array*.

II. Background and Related Works

A. Stochastic Number Generator

One important component in SC is the SNG, which converts an input binary number into the corresponding bit stream. An SNG consists of an RNS and a comparator, as shown in Fig. 2. The RNS generates a random number R in a certain range in each clock cycle. The comparator compares R with the input value X and produces a one if $R<X$ or a zero otherwise. After running for some clock cycles, a bit stream representing the input value is generated. Popular implementations of RNS include linear feedback shift register (LFSR) [9] and Sobol sequence generator [10], [11]. Examples of 3-bit LFSR and Sobol RNSs are shown in Fig. 2.

B. Knobs for Tuning the Accuracy of SC Circuits

For both LFSR and Sobol RNSs, there are some knobs that can be tuned to generate different random number sequences, which in turn affect the accuracy of SC circuits. The LFSR RNS has three knobs: 1) *feedback polynomial*, which describes the subset of DFFs connected to the inputs of the XOR gate[1]; 2) *seed*, which is the initial values stored in the DFFs of an LFSR; 3) *scrambling*, which permutes the LFSR outputs to change the output binary numbers [12]. The Sobol RNS has two knobs: 1) *seed*, which is the initial values stored in the direction vector array; 2) *scrambling*, which is the same as for LFSR [10]. Note that only the feedback polynomial slightly changes the hardware area, while the other knobs have no hardware overhead. The combination of these knobs forms a huge design space. Thus, efficient DSE methods are desired.

C. Accuracy Evaluation

To compare the accuracy of SC circuits with different configurations, two metrics are normally used: *mean absolute error (MAE)* and *mean squared error (MSE)*. For a target univariate function $f(x)$, MAE is calculated as

$$MAE = \frac{1}{2^n}\sum_{x=0}^{2^n-1}\left|f(x)-\tilde{f}(x)\right|, \tag{1}$$

where n denotes the bit width of the input and $\tilde{f}(x)$ denotes the actual output of the SC circuit. MSE is calculated similarly by taking the square value of $(f(x)-\tilde{f}(x))$ instead of the absolute value. The summation in the equations traverses all values of input x, and when generalized to multivariate functions, all possible input combinations are traversed. When the total number of input combinations is too large, we resort to random sampling of input combinations to do the calculation.

D. RNS sharing of SC circuit

RNS occupies a large proportion of area and power of an SC circuit. Alaghi and Hayes proposed a concept called *correlation insensitivity* [13] and showed that if a pair of inputs is *correlation insensitive*, then the RNSs used by them can be shared without affecting the correctness, hence reducing the overall hardware cost. We call this technique *RNS sharing*. The work [13] proposes a method to identify shareable RNSs.

[1]Only some subsets can generate random number sequences with the maximum length.

E. Symbolic Expression of SC circuit

Given that SC is a probabilistic computation, a basic design requirement of an SC circuit is that its output's expectation equals the given target function. Thus, in designing an SC circuit, it is important to derive the symbolic expression of its expected output. For example, the symbolic expression of the output of a two-input AND gate with two independent input probabilities a and b is ab. For simplicity, we refer to the expression as the *symbolic expression of an SC circuit*. The method from [14] can obtain the symbolic expression of a combinational SC circuit. The work [15] refines the method to make it applicable when there are DFFs in the circuit.

III. SCGen for SC Circuits

This section presents SCGen for SC circuits. Its overview is shown in Fig. 3. It provides a C++ interface for users to easily specify SC circuits. To achieve this, we design a wide range of data types corresponding to the basic components of SC circuits, which will be described in Section III-A. How to specify a design in SCGen will be introduced in Section III-B. Once the design is specified, SCGen further supports efficient accuracy simulation, DSE, RNS sharing, symbolic expression analysis, and Verilog generation, which will be described in Sections III-C, III-D, III-E1, III-E2, and III-E3, respectively.

A. SCGen Data Types

Like a general circuit, an SC circuit is typically represented as a graph of nodes and edges. Thus, an SC circuit can be constructed by specifying the nodes and their connections. To ease the node specification, we define various node classes. Fig. 4 shows some classes and their hierarchy. These node classes support the specification of various components of an SC circuit, including LFSR-based and Sobol-based SNGs, basic SC operations like addition, multiplication, and absolute subtraction, and *counters*, which convert stochastic bit streams into binary numbers. How to specify SC circuits using these classes will be introduced in Section III-B. Note that benefiting from abstraction and class hierarchy, SCGen has good extensibility, allowing more components to be added easily.

We use the x^2 SC circuit shown in Fig. 5 to describe some commonly used node classes. The node `DATA` in the figure is an instance of a sub-class of the class `SourceNode`, which determines how the inputs are sampled in calculating accuracy. We provide three sampling methods, corresponding to three sub-classes of `SourceNode`, as shown in Fig. 4. `EnumSourceNode` and `RandSourceNode` correspond to

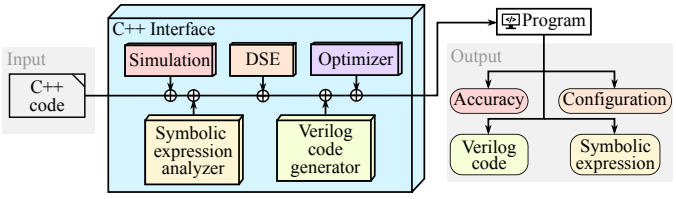

Fig. 3. Overview of SCGen.

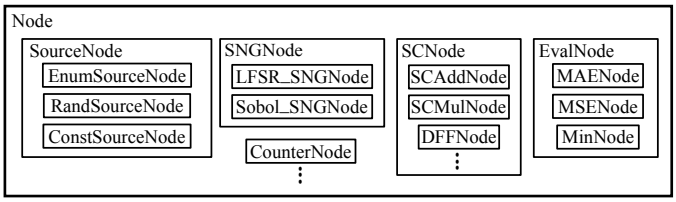

Fig. 4. The hierarchy of the `Node` classes in SCGen.

the traversal and random sampling of input combinations, respectively, as discussed in Section II-C. `ConstSourceNode` outputs a constant number, *e.g.*, the constant 0.5 of the MUX-based SC adder. Nodes `LFSR_SNG`, `DFF`, and `SCMul` in Fig. 5 are instances of the classes `LFSR_SNGNode`, `DFFNode`, and `SCMulNode` shown in Fig. 4, respectively. They represent the LFSR-based SNG, DFF, and the AND gate in the circuit, respectively. The node `Counter` is an instance of the class `CounterNode`, converting a stochastic bit stream to a binary number. The node `MAE` is an instance of the class `MAENode`, which calculates the MAE. The node `MIN` is an instance of the class `MinNode`, which stores the minimum MAE during DSE together with the corresponding configuration.

B. Design Specification

Inspired by Halide [16] and Chisel [17], we develop a C++ interface to provide the users with a high-level abstraction for SC circuit specification. It has the following features:

1) With a high-level abstraction, circuit specification is easier. We define a `Var` class, an abbreviation of "Variable", to represent sub-circuits. Several `Var` instances can be connected to form another `Var` instance. For example, Listing 1 builds the x^2 circuit shown in Fig. 5. By our designed syntax, the second line defines an SC sub-circuit `x` consisting of the `DATA` node and the `LFSR_SNG` node, and the third line connects `x` with DFF and AND gates to form the circuit `res`.

2) SCGen defines a `Conf` class, an abbreviation of "Configuration", to store the global configurations of an SC circuit. For example, the first line in Listing 1 specifies the bit width used in circuit simulation and Verilog generation, and the number of iterations used in accuracy DSE.

3) SCGen overloads operators like "$\star$" and "+" and provides useful functions to build circuits, which are stored in the class `Op`, an abbreviation of "Operation". For example, the third line in Listing 1 inserts a `DFF` node, which adds 1 DFF after `x`, and an `SCMul` node, which multiplies the output of the DFF with `x`. Finally, the `addCalculate` function adds an `MAE` node. By default, a `MIN` is also added for accuracy DSE.

4) Since SCGen is built using C++, it can exploit many C++ features such as loops and standard template library to specify circuits in an efficient way. For example, the code in Listing 2 builds an SC circuit for vector dot product, and the vector dimension can be changed easily by changing the integer `num`.

```
Conf conf(6, 1000); // bit width: 6; DSE times: 1000
Var x(SC, conf, LFSR_SNG);
Var res = x * Op::DFF(x, 1);
res.addCalculate("MAE", [](vector<double> &ins) {
    return pow(ins[0], 2); });
```

Listing 1. Code using C++ interface to build an x^2 SC circuit.

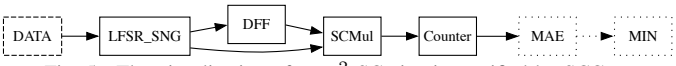

Fig. 5. The visualization of an x^2 SC circuit specified by SCGen.

```
int num = 9;
vector<Var> x(num), y(num), xy(num);
for (int i = 0; i < num; i++) {
     x[i]  = Var(SC, conf, LFSR_SNG);
     y[i]  = Var(SC, conf, LFSR_SNG);
     xy[i] = x[i] * y[i]; }
Var res = Var::sum(xy);
```

Listing 2. Codes using C++ interface to build a vector dot product circuit.

C. Accuracy Simulation

Accuracy simulation is crucial in SC circuit design because it not only indicates the design quality but also plays an important role in accuracy DSE. To improve its efficiency, we propose and implement two acceleration methods in SCGen. Recently, the work [18] utilizes the contingency table to accelerate the SC circuit simulation. However, it gives expectations but not accurate values when inputs are uncorrelated. For example, the accuracy of the SC multiplication circuit with two LFSR SNGs remains unchanged regardless of the LFSR configurations. On the contrary, our proposed methods purely accelerate the simulation, with the same results as the conventional method. In what follows, we illustrate the methods using MAE calculation, but they can be easily adapted for MSE calculation.

1) Incremental Simulation: As shown in Section II-C, the naive method to calculate MAE is to traverse all values of input x and simulate the circuit for each value. However, we notice that when x increases by 1 at a time, for each SNG, only one bit in its output bit stream changes from 0 to 1. Therefore, the simulator only needs to simulate the bit positions with such bit changes, and the other bit positions need not be re-simulated. An example to illustrate this method is the x^2 SC circuit shown in Fig. 6. When input x changes from 6 to 7, the output bit stream of each SNG has only one changed bit, colored in red, and the AND gate only needs to simulate over two bit positions of these two changed bits. This idea can also be extended to simulate SC circuits with DFFs. A DFF node delays the bit stream by 1 clock cycle, so the changed bit will be delayed and affect the next bit position. In this case, we should consider all these bit positions with bit changes.

2) Look-up Table Acceleration: Some sub-circuits of a large SC circuit may have repetitive input patterns during the simulation. In this case, we can store the local outputs of the sub-circuits for different input patterns in a look-up table (LUT) the first time they are encountered, and retrieve the results directly from the LUT when the same input pattern is revisited without re-simulating the sub-circuit. For example, for an SC vector dot product circuit, we can apply this method by maintaining a LUT with two binary inputs and the corresponding output for each multiplication sub-circuit.

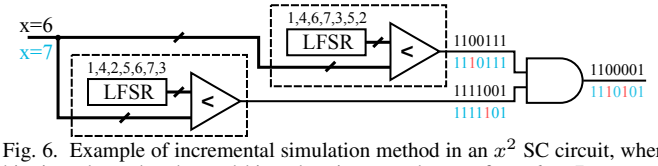

Fig. 6. Example of incremental simulation method in an x^2 SC circuit, where bits in red are the changed bits when input x changes from 6 to 7.

D. Design Space Exploration for Accuracy Maximization

The DSE of SC circuits for accuracy maximization has three steps: the generation of circuit configurations described in Section II-C, the accuracy simulation of each configuration, and the selection of the configuration with the highest accuracy. Besides the basic DSE method that randomly searches configurations, we design two advanced methods based on simulated annealing (SA) and genetic algorithm (GA).

SA and GA are widely-used metaheuristic algorithms for solving optimization problems. Due to space limit, we only highlight their key components used in our context, and we use an SC circuit with 2 LFSRs for illustration. For SA, a key component is the neighbor of a configuration of RNSs. *A configuration of RNSs* for this example with 2 LFSRs is represented by a pair $(LFSR_{conf,1}, LFSR_{conf,2})$, where $LFSR_{conf,l}$ $(l=1,2)$ is the configuration of the l-th LFSR, represented by a 3-tuple (p_i, se_j, sc_k), where p_i is the i-th choice of the polynomial, se_j is the j-th choice of the seed, and sc_k is the k-th choice of the scrambling. Two configurations of RNSs are considered *neighbors* if they only differ by one entry in the configuration of one RNS. For exmaple, $(LFSR_{conf,1}, LFSR_{conf,2})$ and $(LFSR_{conf,1}, LFSR'_{conf,2})$ are neighbors, if $LFSR_{conf,2} = (p_i, se_j, sc_k)$, $LFSR'_{conf,2} = (p_i, se_l, sc_k)$, and $j \neq l$.

For GA, the key components are the *chromosome* and *gene*. The configuration of RNSs is a chromosome, and the configuration of one RNS is a gene. For example, $(LFSR_{conf,1}, LFSR_{conf,2})$ is a chromosome, and $LFSR_{conf,1}$ is a gene. Then, the crossover step in GA swaps the gene sequences of two configurations of RNSs from a random point, and the mutation step randomly changes a gene.

Moreover, the accuracy simulation step can be further accelerated during DSE in addition to the two methods described in Section III-C, by *early termination*. When calculating MAE, each term in the summation is always non-negative, meaning that the summation is non-decreasing. Therefore, when the current partial sum already exceeds the minimum value got so far, the calculation can be terminated immediately, as the final MAE of this configuration will be larger. Note that this method can be combined with the two methods in Section III-C to achieve further speedup.

E. Additional Features of SCGen

This section presents the additional features of SCGen.

1) RNS Sharing Optimizer: SCGen implements the RNS sharing method in [13] to reduce area and power consumption.

2) Symbolic Expression Analyzer: To help check the correctness of a design, SCGen implements a symbolic expression analyzer based on the method in [15], where the symbolic computation is achieved by using GiNaC, a C++ library for symbolic computation [19].

3) Verilog Code Generator: SCGen also supports Verilog generation of the circuit with a test bench, which can be integrated with other codes and used in the following design flow. The generated code automatically utilizes the best circuit configuration from accuracy DSE. Additionally, the generated Verilog modules follow the names in SCGen, which makes the code highly readable and allows further modification.

IV. Extension of SCGen for Hybrid SC-BC Circuits

This section presents the extension of SCGen for hybrid SC-BC circuits. Section III-A introduces the SNG and counter implementations in SCGen, which act as the BC-to-SC conversion and SC-to-BC conversion, respectively. The updates to support hybrid SC-BC circuit design are listed below.

1) For data types, we introduce some classes corresponding to BC operations, such as BC addition and multiplication.

2) We extend the `Var` class in Section III-B to support both SC and BC sub-circuits depending on whether the sub-circuit output is a stochastic bit stream or a binary number.

3) We enhance the operators in the C++ interface so that they can build proper circuits based on the circuit type of the operands. For example, the statement $z = x + y$ connects sub-circuits x and y with an adder. If x and y are SC sub-circuits, the adder is an `SCAddNode`. If they are BC sub-circuits, the adder is a `BCAddNode` instead.

4) The operators also provide automatic conversion between SC and BC sub-circuits. Consider the statement $z = x + y$ again. If x and y are of different types, *e.g.*, x is BC and y is SC, then the type of computation of z is set as that of the left operand, *i.e.*, x. In this case, the sub-circuit y will be automatically converted to BC by inserting a `CounterNode`. Then, a BC adder connects x and y to form z.

5) We extend the automatic RNS sharing algorithm in [13] to support hybrid SC-BC circuits. The extended algorithm is illustrated using a hybrid SC-BC circuit implementing a 2-input neuron shown in Fig. 7a. First, we extract a data flow graph (DFG) from the original circuit as shown in Fig. 7b, where the nodes $a, b, \ldots, e$ represent five LFSR RNSs. The accumulative parallel counter (APC) in Fig. 7a accumulates its two input stochastic bit streams into a binary number and is equivalent to the combination of two counters and a BC adder, as shown in Fig. 7b. Then, we delete the BC part, which is the BC adder, and obtain three independent SC sub-circuits shown in Fig. 7c.

Next, we build a *shareable RNS-pair graph (SRPG)*: Its nodes include all the RNSs, and an edge between nodes i and j exists if and only if the RNSs corresponding to nodes i and j are shareable. To build the shareable RNS-pair graph, we only need to determine all the shareable RNS pairs. To achieve this, for each SC sub-circuit, we apply the method in [13] to identify shareable RNS pairs. Moreover, if two RNSs are in two different SC sub-circuits, they are also shareable. By applying the above rules to the three SC sub-circuits in Fig. 7c, we obtain its corresponding SRPG in Fig. 7d.

Finally, we decide the minimum number of RNSs needed based on the SRPG. If a set of nodes in the SRPG satisfies that each pair of them is connected in the SRPG, they form a *clique*. For example, nodes a, c, and e in Fig. 7d form a clique. Clearly, all the nodes in a clique can share an RNS. Thus, in order to minimize the total number of RNSs, we need to partition all the nodes in the SRPG into the minimum number of cliques. However, finding the minimum number of cliques is NP-hard. In SCGen, we apply a greedy method to obtain an *optimized* clique partition. It first sorts all nodes in the ascending order of their degrees and checks each node in turn to see if it can expand an existing clique. If not, a new

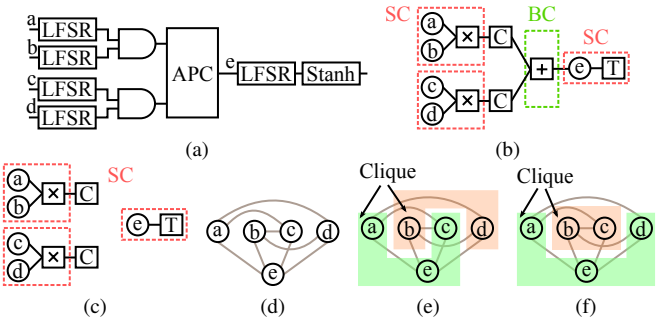

Fig. 7. Illustration of the RNS sharing algorithm for hybrid SC-BC circuits: (a) a hybrid SC-BC circuit implementing a 2-input Neuron, where comparators are omitted for simplicity and Stanh denotes an SC-based tanh module; (b) data flow graph of the 2-input Neuron, where C denotes a counter and T denotes the Stanh module; (c) sub-graphs in SC domain; (d) shareable RNS-pair graph; (e)-(f): two ways of clique partition.

clique is created, which contains that node only. Otherwise, if the node can be assigned to multiple cliques, then a random clique is picked for expansion. However, an optimized clique partition is not unique. Figs. 7e and 7f show two optimized clique partitions for the SRPG in Fig. 7d. Clearly, different optimized clique partitions may lead to different accuracies. Thus, *the choice of the optimized clique partition* can serve as a new knob for accuracy DSE.

V. Experimental Results

This section studies the performance of SCGen. It is evaluated over 10 benchmarks listed in Table I, which covers both small and large designs. The design x^2 is a univariate SC circuit with a single RNS. The designs $x \cdot y$ and $x + y$ are bivariate SC circuits. The rest are real-world applications commonly used in image processing and machine learning [7], including a Roberts cross edge detector, a Sobel kernel, a 2×2 geometric interpolation, GB with different kernel sizes, and artificial neurons with 4 and 8 inputs. Moreover, the GB and artificial neuron circuits are specified as hybrid SC-BC circuits, with the addition implemented in the BC domain. All the experiments are done on a computer with Intel i7-12700F CPU. The hardware performance is measured after synthesis by Synopsys Design Compiler and placement and routing by Cadence Innovus with the Nangate 45nm library.

A. Comparison with Related Works

As introduced in Section I, there are works for SC circuit design and simulation. Their features are summarized in Table II. Compared to them, SCGen is the first to consider the hybrid SC-BC circuit design and support symbolic expression analysis and accuracy DSE.

B. Performance of Acceleration Methods for Accuracy DSE

This section evaluates the acceleration methods for the accuracy DSE. Due to space limit, we only show the results of two typical circuits, where x^2 circuit can be accelerated by incremental simulation and early termination, and GB 3×3 circuit can be accelerated by look-up table and early termination. The runtime of DSE for each circuit with different

Table I. Benchmarks with RNS numbers.

Design	No. of RNS	Design	No. of RNS	Design	No. of RNS	Design	No. of RNS	Design	No. of RNS
x^2	1	$x + y$	3	Sobel	3	GB 3×3	18	Neuron 4	9
$x \cdot y$	2	Roberts	3	Interpolation	6	GB 5×5	50	Neuron 8	17

Table II. Qualitative comparison of SCGen with the related works.

Prior work	scsynth [5]	BitSAD v2 [6]	SCGen (ours)
Language	MATLAB	Julia	C++
Circuit type	SC	SC	Hybrid SC-BC
Accuracy simulation	✗	✓	✓
DSE for accuracy	✗	✗	✓
Hardware optimization	✗	✓	✓
Symbolic expression analysis	✗	✗	✓
Verilog generation	✓	✓	✓

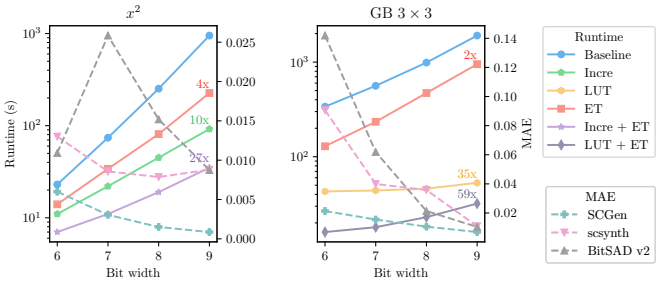

Fig. 8. Comparison of runtimes of different acceleration methods and MAEs of different methods, where "Incre", "LUT", and "ET" stand for incremental simulation, look-up table method, and early termination, respectively.

bit widths and acceleration methods is shown in Fig. 8, and the numbers are the speed-up ratios at bit width of 9 over the baseline. The results show that the acceleration methods for accuracy DSE can achieve up to $59\times$ speedup. Fig. 8 also compares the accuracy of the generated circuits with existing works scsynth [5] and BitSAD v2 [6], which do not perform accuracy DSE. The results show that the accuracy DSE is essential to push the accuracy limit of SC circuits.

C. Performance of DSE for Accuracy Maximization

This section evaluates the SA and GA methods described in Section III-D, with random search as the baseline. We set the bit width to 6 and the DSE iteration number to 1000. For simulations in DSE, 10000 pairs of random input patterns are generated. For each circuit, we run all three DSE methods 50 times and record the minimum MAE found. Fig. 9 shows the results on all benchmarks, with MAE ratios of SA and GA relative to the random search on top. The results show that SA and GA methods reduce MAE by 4.0% and 12.7%, respectively. Besides, the SA method performs better on small circuits, while the GA method is better on larger circuits.

D. Performance of RNS Sharing

This section studies the performance of RNS sharing. We first evaluate the effect of RNS sharing on hardware cost reduction. The bit width is 8, the clock period is 20ns, and LFSR is used as RNS. Among all benchmarks, the benchmarks x^2, $x \cdot y$, Roberts, and Sobel cannot be optimized by RNS

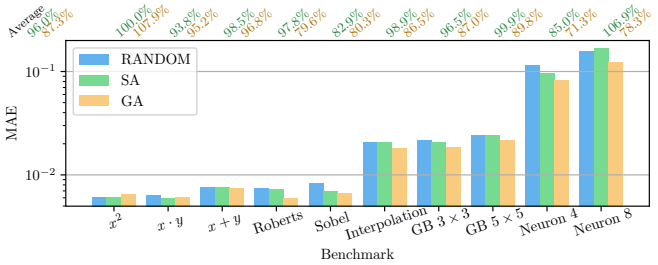

Fig. 9. Minimum MAE found by different DSE methods, with ratios of SA and GA relative to random search on top and the average ratios over all the benchmarks in the top-left corner.

Table III. Hardware metrics of circuits before and after RNS sharing with the improvement in area and power shown in parentheses.

Application	Shareable RNS Pairs	No. of LFSR before sharing	No. of LFSR after sharing	Area (μm^2) before sharing	Area (μm^2) after sharing	Power (mW) before sharing	Power (mW) after sharing	SCGen code length	Verilog code length (core module)	Verilog code length (testbench)
$x + y$	2	3	2	318.67	268.13 (15.86%)	3.37e-5	2.80e-5 (16.91%)	21	82	46
Interpolation	6	6	3	589.72	431.45 (26.84%)	6.25e-5	4.34e-5 (30.56%)	31	151	58
GB 3×3	144	18	2	1647.34	771.67 (53.16%)	1.77e-4	6.26e-5 (64.63%)	32	385	67
GB 5×5	1200	50	2	4479.97	1822.63 (59.32%)	4.86e-4	1.49e-4 (69.34%)	40	1025	115
Neuron 4	32	9	2	928.34	547.96 (40.97%)	1.02e-4	5.35e-5 (47.55%)	33	219	52
Neuron 8	128	17	2	1605.04	793.74 (50.55%)	1.80e-4	7.67e-5 (57.39%)	35	379	64
Average				1594.85	772.59 (41.12%)	1.74e-4	6.89e-5 (47.73%)	32	373.5	67

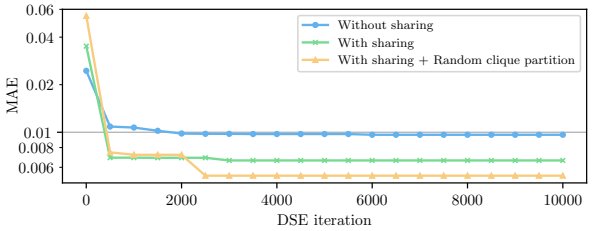

Fig. 10. Effect on the accuracy of RNS sharing and considering different clique partitions as a new dimension in DSE.

sharing, and hence, are excluded. The experimental results are shown in Table III, where the improvement in area and power is put inside the parentheses. The results show that RNS sharing reduces area and power by 41.1% and 47.7% on average, respectively. The RNS sharing using Sobol RNS achieves similar performance and is omitted due to space limit. The phenomenon of shareable RNS pairs is prevailing in many applications. The effect on hardware cost reduction is more significant for large circuits.

Then, we study the effect of our RNS sharing algorithm on accuracy. We simulate the GB 3×3 circuit under different numbers of DSE iterations. The result is shown in Fig. 10. Note that our RNS sharing algorithm introduces another dimension for DSE, *i.e.*, the choice of clique partitioning. The green and yellow lines correspond to the results of using RNS sharing without and with considering different choices of clique partitions, respectively. From Fig. 10, we can see that RNS sharing can achieve higher accuracy given the same DSE times by removing irrelevant points in the design space. Moreover, considering different clique partitions as a new dimension in DSE can further improve the accuracy.

E. Development Efficiency of Using SCGen

To demonstrate the development efficiency of using SCGen, we measure the code length of circuits specified in SCGen and in Verilog as shown in Table III. The results show that using SCGen can reduce code length by $11\times$ compared to using Verilog. Therefore, designing hybrid SC-BC circuits via SCGen can significantly reduce the amount of work required.

F. Case Study: Gaussian Blur

We further test the 8-bit GB 3×3 circuit on a real image to find out the improvement gained from the RNS sharing and the accelerated DSE for accuracy maximization. The best configuration found from DSE is used to build the circuit to generate the output image. The quality is then quantified by the peak signal-to-noise ratio (PSNR). The objective of the accuracy DSE is MSE since it is more relevant to the PSNR calculation. The experimental results are shown in Table IV. The results show that our proposed method can do more DSE iterations with less runtime and is able to find a circuit configuration with higher accuracy.

Table IV. Comparison of the DSE performance on GB 3×3 circuit.

Method	# of DSE iterations	Runtime (s)	Minimal MSE	PSNR (dB)
Baseline	100	715	0.000184	30.05
Proposed	**2000**	**343**	**0.000049**	**40.34**

VI. Conclusions and Future Works

In this paper, we propose SCGen, a versatile generator framework for agile design of stochastic circuits. It provides a powerful C++ interface to specify SC circuits and is able to do optimization, simulation, accelerated DSE for accuracy maximization, validation, and Verilog code generation. The experiment results demonstrate the effectiveness of SCGen. In addition, SCGen is designed with good extensibility. Thus, it is easy to enhance its features. Our future work will extend SCGen to support more effective DSE.

References

[1] A. Alaghi and J. P. Hayes, "Survey of stochastic computing," *ACM TECS*, vol. 12, no. 2s, pp. 1–19, 2013.
[2] B. R. Gaines, "Stochastic computing," in *AFIPS*, 1967, pp. 149–156.
[3] A. Alaghi *et al.*, "Stochastic circuits for real-time image-processing applications," in *DAC*, 2013, pp. 1–6.
[4] Y. Zhang *et al.*, "When sorting network meets parallel bitstreams: A fault-tolerant parallel ternary neural network accelerator based on stochastic computing," in *DATE*, 2020, pp. 1287–1290.
[5] A. Alaghi, "scsynth," https://github.com/arminalaghi/scsynth, 2016.
[6] K. Daruwalla *et al.*, "Bitsad v2: Compiler optimization and analysis for bitstream computing," *ACM TACO*, vol. 16, pp. 1–25, 2019.
[7] V. T. Lee *et al.*, "Architecture considerations for stochastic computing accelerators," *IEEE TCAD*, vol. 37, no. 11, pp. 2277–2289, 2018.
[8] M. H. Sadi and A. Mahani, "Accelerating deep convolutional neural network base on stochastic computing," *Integration*, vol. 76, pp. 113–121, 2021.
[9] S. Golomb, *Shift Register Sequences*. Laguna Hills, CA, USA: Aegean Park Press, 1982.
[10] P. Bratley and B. L. Fox, "Algorithm 659: Implementing Sobol's quasirandom sequence generator," *ACM TOMS*, vol. 14, no. 1, pp. 88–100, 1988.
[11] S. Liu and J. Han, "Energy efficient stochastic computing with Sobol sequences," in *DATE*, 2017, pp. 650–653.
[12] J. H. Anderson *et al.*, "Effect of LFSR seeding, scrambling and feedback polynomial on stochastic computing accuracy," in *DATE*, 2016, pp. 1550–1555.
[13] A. Alaghi and J. P. Hayes, "Dimension reduction in statistical simulation of digital circuits," in *DEVS*, 2015, pp. 1–8.
[14] K. Parker and E. Mccluskey, "Probabilistic treatment of general combinational networks," *IEEE TOC*, vol. C-24, no. 6, pp. 668–670, 1975.
[15] Z. Li *et al.*, "Simultaneous area and latency optimization for stochastic circuits by D flip-flop insertion," *IEEE TCAD*, vol. 38, no. 7, pp. 1251–1264, 2019.
[16] J. Ragan-Kelley *et al.*, "Decoupling algorithms from schedules for easy optimization of image processing pipelines," *ACM TOG*, vol. 31, no. 4, pp. 1–12, 2012.
[17] J. Bachrach *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *DAC*, 2012, pp. 1212–1221.
[18] S. Aygun *et al.*, "Bit-stream processing with no bit-stream: Efficient software simulation of stochastic vision machines," in *GLSVLSI*, 2023, pp. 273–279.
[19] J. Vollinga, "GiNaC–symbolic computation with c++," *NIM-A*, vol. 559, no. 1, pp. 282–284, 2006.