Towards Low-Cost High-Accuracy Stochastic Computing Architecture for Univariate Functions: Design and Design Space Exploration

Kuncai Zhong[1], Zexi Li[1], Weikang Qian[1,2,*]
[1]University of Michigan-Shanghai Jiao Tong University Joint Institute, Shanghai Jiao Tong University, China
[2]MoE Key Laboratory of Artificial Intelligence, Shanghai Jiao Tong University, China
Email: kczhong@sjtu.edu.cn, lzx12138@sjtu.edu.cn, qianwk@sjtu.edu.cn; *corresponding author

***Abstract*—Univariate functions are widely used. Several recent works propose to implement them by an unconventional computing paradigm, stochastic computing (SC). However, existing SC designs either have a high hardware cost due to the area-consuming randomizer or a low accuracy. In this work, we propose a low-cost high-accuracy SC architecture for univariate functions. It consists of only a single stochastic number generator and a minimum number of D flip-flops. We also apply three methods, random number source (RNS) negating, RNS scrambling, and input scrambling, to improve the accuracy of the architecture. To efficiently configure the architecture to achieve a high accuracy, we further propose a design space exploration algorithm. The experimental results show that compared to the conventional architecture, the area of the proposed architecture is reduced by up to 76%, while its accuracy is close to or sometimes even higher than that of the conventional architecture.**

I. Introduction

Univariate functions such as sigmoid and hyperbolic tangent functions are widely used [1]. Recent studies proposed several implementations for univariate functions by an unconventional computing paradigm, *stochastic computing (SC)* [2]. SC computes on *stochastic numbers (SNs)*. An SN is a sequence of bits that represents a value by the ratio of 1s in the sequence. For example, an SN 01001100 represents the value $\frac{3}{8}$. SC circuits usually have low hardware cost and strong fault tolerance compared to the conventional binary computing [3]. Therefore, SC attracts much attention recently [4]. In this work, we focus on designing SC circuits for univariate functions. For simplicity, we call such circuits *univariate SC circuits* and their architectures *univariate SC architectures*.

A univariate SC circuit consists of a *randomizer* and an *SC core*. The randomizer converts the binary radix-encoded variable X to SNs and may optionally provide some SNs of constant values. The SC core is the core to implement the function. It takes as input the SNs provided by the randomizer and computes the final output SN. Among the existing SC cores for univariate functions, the one proposed in [5] has low hardware cost. Thus, we also use it in our architecture. As shown in Fig. 1, it is a combinational circuit, taking as input d independent SNs of the same value x (hereafter referred to as *variable SNs*) and m independent SNs of the same value 0.5 (hereafter referred to as 0.5 *SNs*). A conventional randomizer to generate the $(d+m)$ SNs is also shown in Fig. 1. It includes d stochastic number generators (SNGs) to generate d variable SNs. An SNG consists of a random number source (RNS) and a comparator (CMP), where an n-bit RNS generates n random bits with probability of 0.5 being a one in each clock. Examples of RNS include linear feedback shift register (LFSR) and Sobol sequence generator (SSG) [6]. As shown in Fig. 1, the randomizer has an extra m-bit RNS to provide m 0.5 SNs to the SC core. We call the architecture in Fig. 1 *the conventional univariate SC architecture*.

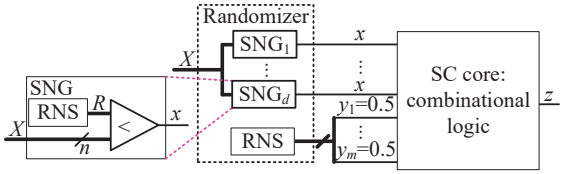
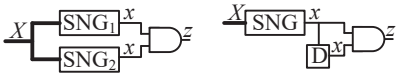


Fig. 1. The conventional univariate SC architecture.

For the conventional architecture, the randomizer generally occupies most area of an SC circuit. For example, for the SC squarer in Fig. 2(a), with 8-bit SSGs as the RNSs, the randomizer can even occupy 99.7% area. Hence, to reduce the hardware cost, it is critical to optimize the randomizer. Several methods have been proposed for this purpose, such as RNS scrambling [7], [8] and inserting D flip-flops (DFFs) [9]–[11]. However, these methods will also decrease the accuracy of the circuit. For example, for the SC squarer in Fig. 2(a), we can share an SNG for the 2 inputs and then insert one DFF as shown in Fig. 2(b). Clearly, the circuit area reduces. Nevertheless, with 8-bit SSGs as the RNSs, the mean absolute error (MAE) increases from 0.0042 to 0.0873, which is calculated as follows:

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

where $f(X)$ and $f'(X)$ are the expected and the real outputs of a univariate SC circuit with X as the input, respectively. Therefore, how to design a univariate SC architecture with low hardware cost and high accuracy remains as an open problem.



(a) (b)

Fig. 2. SC square circuits (a) with 2 SNGs and (b) with an SNG shared and a DFF inserted.

In this work, we try to address the above problem. Our contributions are as follows.

- We propose a *basic univariate SC architecture*, which only needs a single RNS, a single CMP, and $(d-1)$ DFFs to generate $(d+m)$ input SNs (see Section III). Compared to the conventional architecture, the basic architecture has a dramatically reduced area.
- To improve the accuracy, we further apply three methods, RNS negating, RNS scrambling, and input scrambling, to

the basic architecture, and obtain an *improved univariate SC architecture* (see Section IV). These three methods cost no hardware overhead. Thus, the improved architecture has the same area as the basic architecture.
- As the accuracy of the improved architecture is determined by the choice of RNS scrambling, RNS negating, and input scrambling, we propose an algorithm to decide a good choice giving a high accuracy (see Section V).

The experimental results show that compared to the conventional univariate SC architecture, the area of the improved univariate SC architecture is reduced by up to 76%, while its accuracy is close to or sometimes even better than that of the conventional architecture (see Section VI).

II. Related Works

In this section, we present related works on randomizer optimization. They can be classified into three categories:

(1) RNS optimization: RNS optimization is to design better RNSs for SC to achieve high accuracy or low hardware cost. A typical example is SSG [6]. The SN generated by it has low discrepancy [12], and an SC circuit with SSGs as RNSs generally has high accuracy. Another example is SBoNG, which can simultaneously generate multiple SNs with a low hardware cost [13]. These optimized RNSs can be directly applied to our proposed univariate SC architecture.

(2) RNS scrambling: Scrambling reorders the input signals, as shown in Fig. 3. RNS scrambling reorders the output bits of an RNS to generate a different random binary number, and hence leads to different SNs. This may improve the accuracy of an SC circuit. Anderson *et al.* applied it to *each* RNS in an SC circuit with multiple RNSs to improve the output accuracy [7]. However, their method does not reduce the hardware cost of the randomizer. To reduce the hardware cost, Salehi applied multiple different RNS scramblings to a single RNS for generating multiple SNs with low correlation [8]. However, the randomizer from [8] still needs multiple CMPs for generating multiple SNs. As we show in our work, for a univariate function, the CMP number can be reduced to 1.

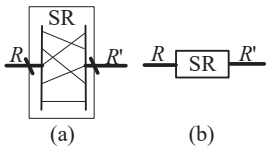


Fig. 3. (a) scrambling and (b) its symbol.

(3) DFFs insertion: DFF insertion is a method to delay an SN for some clock cycles by inserting some DFFs [9]. If the bits of the SN are independently generated, DFF insertion can generate an SN of the same value as the original one, but uncorrelated to it. Fig. 2(b) shows an example of implementing the function $f(x) = x^2$ by DFF insertion with reduced randomizer area. To minimize the number of DFFs inserted, Ting and Hayes formulated an integer linear programming (ILP) problem [10]. Li *et al.* further proposed an ILP formulation to exploit DFF insertion to simultaneously reduce area and delay of an SC circuit [11]. However, these works only focus on the hardware cost reduction by DFF insertion and do not consider their influence to accuracy.

III. Basic Univariate SC Architecture

The proposed basic univariate SC architecture is shown in Fig. 4. Its SC core is the one from [5]. Its randomizer consists of an RNS, a CMP, a bit selection module, and $(d-1)$ DFFs.

In order to generate d mutually uncorrelated variable SNs of the same value x, we first use a single SNG consisting of an RNS and a CMP to generate a variable SN of the value x. Then, we insert $(d-1)$ DFFs to delay this variable SN for $1, 2, \ldots, d-1$ clock cycles to generate the other $(d-1)$ variable SNs, as shown in Fig. 4. The bit width n of the RNS is set to be at least m. To generate the m 0.5 SNs, a bit selection module is applied to choose m outputs from the n outputs of the RNS. Note that the module is logic-free; it is implemented by direct wire connection.

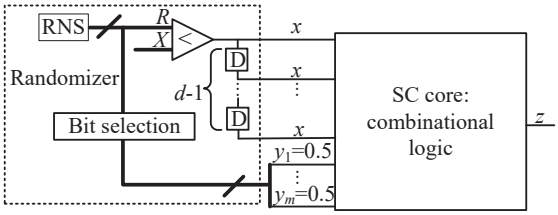


Fig. 4. The basic univariate SC architecture.

Clearly, the numbers of RNSs and CMPs in the basic architecture reach the *minimum* value. Besides, the number of DFFs inserted also reaches the minimum. We have the following claim.

Claim 1 *To generate d mutually uncorrelated SNs of the same value using a single SNG, we need to insert a minimum of $(d-1)$ DFFs.*

Proof: First, since DFF insertion can produce mutually uncorrelated SNs of the same value, the SNG together with the $(d-1)$ DFFs shown in Fig. 4 can generate d mutually uncorrelated SNs of the same value.

Now assume that we can use $s < d-1$ DFFs and a single SNG to generate d mutually uncorrelated SNs of the same value. Let the SN generated by the SNG be x^*. Suppose the $(d-1)$ SNs generated by the s DFFs are x^* delayed by $s_1, s_2, \ldots, s_{d-1}$ clock cycles, where $s_1, \ldots, s_{d-1}$ are positive integers. Since $s < d-1$, we can easily see that $0 < s_1, \ldots, s_{d-1} \leq s < d-1$. Thus, there exist $1 \leq i, j \leq d-1$ such that $i \neq j$ and $s_i = s_j$. This means that there exist two SNs that are delayed by the same number of clock cycles from x^*. Thus, they are not uncorrelated. Consequently, the initial assumption is incorrect. Therefore, the minimum number of DFFs needed is $(d-1)$. □

Compared to the conventional univariate SC architecture, the basic architecture has a dramatically reduced hardware cost. However, its accuracy is usually not high. We need to further improve its accuracy.

IV. Improved Univariate SC Architecture

In this section, we first describe three methods to improve the accuracy of the basic univariate SC architecture, which are RNS negating, RNS scrambling, and input scrambling. Then, by applying them, we derive an improved univariate SC architecture.

A. Three methods for improving accuracy

(1) RNS negating: RNS negating is shown in Fig. 5. It negates some output bits of an RNS. When we apply it to the basic univariate SC architecture, the output order of the random binary numbers produced by the RNS changes. Consequently, the order of the bits in the SN generated by the RNS changes, which may improve the output accuracy of the SC circuit. Since a DFF simultaneously outputs a signal and its complement, and LFSRs and SSGs use DFFs to output random binary numbers, thus, if we use an LFSR or an SSG as the RNS, RNS negating has no additional hardware cost.

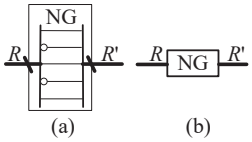


Fig. 5. (a) RNS negating and (b) its symbol.

(2) RNS scrambling: As introduced in Section II, RNS scrambling permutes the output bits of an RNS and may improve the accuracy of the SC circuit. In our case, we apply it to the basic univariate SC architecture. Same as RNS negating, it has no area overhead.

(3) Input scrambling: As shown in Fig. 4, the randomizer in the basic univariate SC architecture generates multiple variable SNs of the same value, which are connected to the corresponding inputs of the SC core. By scrambling the connection order, the output accuracy of the circuit may improve. For example, consider an instance of the basic univariate SC architecture shown in Fig. 6(a), which implements $f(x) = x + x^2 - x^3$ and has 3 variable SNs. A scrambling module is inserted between the outputs of the randomizer and the inputs of the SC core. There are $3! = 6$ different scrambling ways, corresponding to 6 different connections. The MAEs of the circuits with these scrambling ways applied are shown in Fig. 6(b), where an 8-bit LFSR is used as the RNS. Clearly, the MAE varies with different scrambling ways, so a proper choice may improve the accuracy. We call the particular scrambling *input scrambling*, which differs from the RNS scrambling. Note that it also has no area overhead.

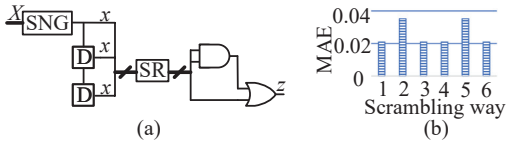


Fig. 6. (a): A univariate SC circuit implementing $f(x) = x + x^2 - x^3$; (b): MAEs for the circuit in (a) with different scrambling ways.

B. Improved univariate SC architecture

By applying the above three accuracy-improvement methods to the basic univariate SC architecture, we eventually propose an improved univariate SC architecture as shown in Fig. 7.

To generate the d variable SNs, we insert an RNS negating module NG_1 followed by an RNS scrambling module SR_1 after the RNS. To generate the m 0.5 SNs, we insert an RNS negating module NG_2 followed by an RNS scrambling module SR_2 after the bit selection module BS. Finally, we insert an input scrambling module SR_3 to further improve the accuracy.

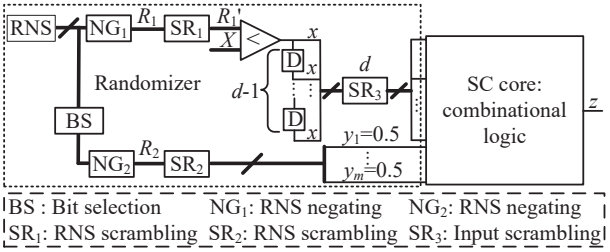


Fig. 7. The improved univariate SC architecture.

Note that the extra modules added have no area overhead. Thus, the improved univariate SC architecture has the same area as the basic one. Compared to the conventional univariate SC architecture, the improved one has a much reduced area.

V. Design Space Exploration for the Improved Univariate SC Architecture

When we apply the improved univariate SC architecture to implement a given target function, we need to determine a proper configuration of the modules RNS, BS, NG_1, NG_2, SR_1, SR_2, SR_3, and the SC core in Fig. 7 to minimize the output error. We assume that the SC core is synthesized by the method in [14], so it is determined. Thus, we need to determine the configurations of the remaining 7 modules. All the possible configurations of the 7 modules form a large design space for the improved univariate SC architecture. In this section, we first characterize the design space and then show a design space exploration algorithm to reach a good configuration.

A. Design space characterization

We first introduce some definitions as follows.

1) We define the set of possible selections of m bits from an n-bit RNS as B, and the ith selection as B_i. The size of B is $\binom{n}{m}$. B_1 denotes the selection of the 1st to the mth bits of the RNS.
2) We define the set of possible permutations of k bits as R^k, and the ith permutation as R^k_i. The size of R^k is $k!$. We use R^k_1 and $R^k_{k!}$ to denote the reverse order and the original order of the k bits, respectively.
3) We define the set of possible negations of a subset of k bits as C^k, and the ith negation as C^k_i. The size of C^k is 2^k. $C^k_{2^k}$ denotes the negation of all the k bits.
4) Given a specific RNS type, there also exist multiple detailed configurations for it. For example, an LFSR has multiple choices for its feedback polynomial. We define the set of given RNS configurations as L, the size of L as l, and the ith configuration in L as L_i.

Based on the above definitions and analysis, we can directly list the number of choices for each module of the improved univariate SC architecture in Table I.

Thus, the size of the entire design space is $l\frac{(n!)^2 d!}{(n-m)!}2^{n+m}$, which is very large when n or m is large. On the one hand, the large design space is *beneficial* since it is more likely to contain a design point with a low output error. On the other

hand, it poses a challenge to exhaustively search every possible point in the design space. In the following, we propose an algorithm to efficiently explore the design space.

TABLE I
NUMBER OF CHOICES FOR EACH MODULE OF THE IMPROVED UNIVARIATE SC ARCHITECTURE.

Module	RNS	BS	NG_1	NG_2	SR_1	SR_2	SR_3
Number	l	$\binom{n}{m}$	2^n	2^m	$n!$	$m!$	$d!$

B. Proposed algorithm

The proposed algorithm iteratively searches proper configurations for some modules while keeping those of the others fixed. We first introduce the subroutines that configure some modules in the improved univariate SC architecture. Then, by combining them together, we present the whole algorithm.

In these subroutines, *calMAE* is a function to calculate the MAE of the improved univariate SC architecture. It has 7 inputs, which are the configurations of 7 modules, RNS, BS, NG_1, NG_2, SR_1, SR_2, and SR_3. Furthermore, RNS*, BS*, NG_1^*, NG_2^*, SR_1^*, SR_2^*, and SR_3^* denote the running best choices we have found for the 7 modules, respectively. MAE_{min} denotes the running minimum MAE.

Algorithm 1 shows the first subroutine that configures RNS, BS, NG_1, and NG_2. It exhaustively searches all the combinations of choices for BS, NG_1 and NG_2, and randomly picks a choice for RNS. The modules SR_1, SR_2, and SR_3 are fixed as their current best choices. In the process, if we obtain a smaller MAE, we update the best choices for RNS, BS, NG_1, and NG_2.

Algorithm 1: Configuring RNS, BS, NG_1, and NG_2.

```
1  input: MAE_min, RNS*, BS*, NG1*, NG2*, SR1*, SR2*, and SR3*;
2  output: MAE_min, RNS*, BS*, NG1*, NG2*;
3  foreach B_i in B do
4      foreach C_j^n in C^n do
5          foreach C_k^m in C^m do
6              generate a random number 1 ≤ r ≤ l;
7              MAE←calMAE(L_r,B_i,C_j^n,C_k^m,SR1*,SR2*,SR3*);
8              if MAE<MAE_min then
9                  MAE_min ←MAE; RNS* ← L_r; BS* ← B_i;
                   NG1* ← C_j^n; NG2* ← C_k^m;
10 return MAE_min, RNS*,BS*,NG1*,NG2*;
```

Algorithm 2 shows the second subroutine that configures RNS, SR_1, and SR_2. It has two loops. In the first loop, it exhaustively searches all the choices for SR_1, and randomly selects choices for RNS and SR_2. In the second loop, it exhaustively searches all the choices for SR_2, and randomly selects choices for RNS and SR_1. The modules BS, NG_1, NG_2, and SR_3 are fixed as their current best choices. In these two loops, when we obtain a smaller MAE, we update the best choices for RNS, SR_1, and SR_2.

Algorithm 3 shows the third subroutine that configures SR_3. It exhaustively searches all the choices for SR_3, while setting the remaining modules with their current best choices. When a smaller MAE is found, it updates the best choice for SR_3.

With the previous subroutines, we finally propose the whole algorithm as shown in Algorithm 4. Line 3 initializes RNS*, BS*, NG_1^*, NG_2^*, SR_1^*, SR_2^*, and SR_3^* as L_1, B_1, $C_{2^n}^n$, $C_{2^m}^m$, R_1^n, $R_{m!}^m$, and $R_{d!}^d$, respectively. Based on these initial choices, Line 4 calculates the initial MAE_{min}. To improve the accuracy, Line 7 first updates the best choices for RNS, BS, NG_1, and NG_2 by Algorithm 1. Then, Line 8 updates the best choices for RNS, SR_1, and SR_2 by Algorithm 2. Finally, Line 9 updates the best choice for SR_3 by Algorithm 3. We repeat the whole process until the current iteration does not improve MAE_{min} over the previous one. In general, the number of iterations is less than 10. Therefore, the time complexity is dominated by that of Algorithms 1, 2, and 3, which are $O(\frac{n!}{m!(n-m)!}2^{n+m})$, $O(n!+m!)$, and $O(d!)$, respectively. Thus, the time complexity of the proposed algorithm is $O(\frac{n!}{m!(n-m)!}2^{n+m}+n!+m!+d!)$, which is far smaller than the design space size $l\frac{(n!)^2d!}{(n-m)!}2^{n+m}$.

Algorithm 2: Configuring RNS, SR_1, and SR_2.

```
1  input: MAE_min, RNS*, BS*, NG1*, NG2*, SR1*, SR2*, and SR3*;
2  output: MAE_min, RNS*, SR1*, SR2*;
3  foreach R_i^n in R^n do
4      generate a random number 1 ≤ r ≤ l;
5      generate a random number 1 ≤ k ≤ m!;
6      MAE←calMAE(L_r,BS*,NG1*,NG2*,R_i^n,R_k^m,SR3*);
7      if MAE<MAE_min then
8          MAE_min ←MAE; RNS* ← L_r; SR1* ← R_i^n; SR2* ← R_k^m;
9  foreach R_i^m in R^m do
10     generate a random number 1 ≤ r ≤ l;
11     generate a random number 1 ≤ k ≤ n!;
12     MAE←calMAE(L_r,BS*,NG1*,NG2*,R_k^n,R_i^m,SR3*);
13     if MAE<MAE_min then
14         MAE_min ←MAE; RNS* ← L_r; SR1* ← R_k^n; SR2* ← R_i^m;
15 return MAE_min, RNS*, SR1*, SR2*;
```

Algorithm 3: Configuring SR_3.

```
1  input: MAE_min, RNS*, BS*, NG1*, NG2*, SR1*, SR2*, and SR3*;
2  output: MAE_min, SR3*;
3  foreach R_i^d in R^d do
4      MAE←calMAE(RNS*,BS*,NG1*,NG2*,SR1*,SR2*,R_i^d);
5      if MAE<MAE_min then MAE_min ←MAE; SR3* ← R_i^d;
6  return MAE_min, SR3*;
```

Algorithm 4: The proposed design space exploration algorithm.

```
1  input: an improved univariate SC architecture for a target function;
2  output: MAE_min, RNS*, BS*, NG1*, NG2*, SR1*, SR2*, and SR3*;
3  RNS* ← L_1; BS* ← B_1; NG1* ← C_{2^n}^n; NG2* ← C_{2^m}^m; SR1* ← R_1^n;
     SR2* ← R_{m!}^m; SR3* ← R_{d!}^d;
4  MAE_min ←calMAE(RNS*,BS*,NG1*,NG2*,SR1*,SR2*,SR3*);
5  MAE_prev ← MAE_min;
6  while true do
7      update MAE_min, RNS*, BS*,NG1*, NG2* by Algorithm 1;
8      update MAE_min, RNS*, SR1*, SR2* by Algorithm 2;
9      update MAE_min, SR3* by Algorithm 3;
10     if MAE_min < MAE_prev then MAE_prev ← MAE_min;
11     else break;
12 return MAE_min, RNS*, BS*, NG1*, NG2*, SR1*, SR2*, and SR3*;
```

VI. EXPERIMENTAL RESULTS

In this section, we show the experimental results. We evaluate the accuracy and the hardware cost of the improved univariate SC architecture.

A. Experimental setup

We choose 12 univariate functions as the test cases, which are listed in Table II together with their IDs. For each function, we apply the method proposed in [14] to synthesize the SC core, which is used in the univariate SC architecture. We set $d = 4$ and $m = 6$. Thus, each circuit needs 4 variable SNs and 6 0.5 SNs.

We compare the improved univariate SC architecture with 2 other architectures. The first is the conventional univariate SC architecture shown in Fig. 1. Its randomizer consists of d n-bit SNGs and an m-bit RNS. The second is a state-of-the-art SC architecture proposed in [8]. It applies RNS scrambling to a single n-bit RNS $(d+1)$ times for generating d variable SNs and m 0.5 SNs. Its randomizer consists of a single n-bit RNS and d CMPs. We call it *scrambling univariate SC architecture*.

TABLE II
THE TARGET UNIVARIATE FUNCTIONS USED IN OUR EXPERIMENTS.

ID	Function	ID	Function	ID	Function	ID	Function
1	$\sin(x)$	4	$\log(1+x)$	7	$\tanh(4x)$	10	$\frac{1}{1+e^{-x}}$
2	$\cos(x)$	5	$\frac{\sin(\pi x)}{\pi}$	8	$x^{0.45}$	11	$x^{2.2}$
3	e^{-x}	6	$\tanh(x)$	9	e^{-2x}	12	$0.5\cos(\pi x)+0.5$

For the univariate SC architectures, we choose both LFSR and SSG as the RNSs. Since our architecture can benefit from SSG by directly using it as the RNS, we do not compare our architecture with other SSG-based designs. In the following, we denote the conventional architecture, the scrambling architecture, and the improved architecture based on LFSRs as *CL, SL, IL**, respectively, and the conventional architecture, the scrambling architecture, and the improved architecture based on SSGs as *CS, SS, IS**, respectively.[1] Note that SC is not applicable to high-precision computation due to its long latency. Thus, in the following, we compare these architectures for RNS bit width $n = 8$.[2]

B. Accuracy comparison

In this section, we compare accuracies of the conventional, the scrambling, and the improved univariate SC architectures. We use MAE as the accuracy measure.

To achieve a high accuracy for the conventional and the scrambling architectures, we randomly generate a number of configurations for their randomizer and choose the one with the lowest MAE. For both architectures, their randomizer configuration includes the RNS configuration (i.e., feedback polynomials for LFSRs and direction vectors for SSGs [6]) and the RNS scrambling ways. Furthermore, for the conventional architecture using LFSRs, the randomizer configuration also includes the LFSR seeding, which is another method to improve the accuracy [7]. For the scrambling architecture, the randomizer configuration also includes the selection of m bits from the only RNS to generate the m 0.5 SNs.

For the improved univariate SC architecture, we apply our proposed algorithm to find a high-accuracy configuration. For a fair comparison to the improved architecture with a specific RNS type configured by our algorithm, the runtime of the random search for the conventional and the scrambling architectures with the same RNS type is set the same as our algorithm.

Note that an SC core synthesized by [14] implements a polynomial approximating a target non-polynomial function. Thus, there exists an approximation error between the target function and the actual function realized by the SC core. We define the mean approximation error as the *baseline MAE*. The normalized MAEs to the baseline MAE for various architectures, RNS types, and target functions are plotted in Fig. 8. We list the baseline MAEs at the top of the figure.

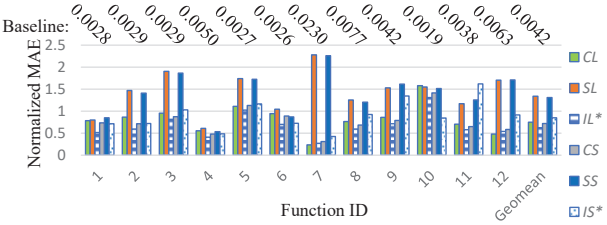


Fig. 8. Normalized MAE comparison for different SC architectures.

For an SC circuit, its entire error consists of two components, the approximation error and the stochastic error due to randomizer [15]. The entire error is usually larger than the approximation error. However, as shown in the figure, the normalized MAEs of *IL** and *IS** are usually much smaller than 1. This indicates that through our proposed design space exploration method, the stochastic error can even cancel out the approximation error, leading to a decrease in the entire error. Furthermore, for most test cases, the MAE of *IL** is smaller than that of *CL* and *SL*. Thus, when we apply LFSRs as the RNS, the improved architecture generally has a higher accuracy than the conventional and the scrambling architectures. Moreover, the MAE of *IS** is smaller than that of *CS* for one-fourth of the test cases, and is smaller than that of *SS* for most test cases. Therefore, when we apply SSGs as the RNS, the accuracy of the improved architecture is sometimes higher than that of the conventional and the scrambling ones.

C. Hardware cost comparison

We first compare the conventional, the scrambling, and the improved univariate SC architectures in terms of area. We derive the area of an SC circuit by summing up the areas of its components. For example, to obtain the area of the improved univariate SC architecture for $\sin(x)$, we sum up the areas of an RNS, a CMP, 3 DFFs and the SC core for $\sin(x)$. Table III lists the area of each component, which is obtained by Synopsys Design Compiler [16] using the Nangate 45nm library [17]. The area of each function is the area of the corresponding SC core. Note that the 6-bit LFSR and the 6-bit SSG in the table are used in the conventional univariate SC architecture for generating 6 0.5 SNs.

Fig. 9 compares the areas of different architectures for different functions and RNS types. Compared to the conventional architecture *CL* (resp. *CS*), the improved architecture *IL** (resp. *IS**) can reduce 70% (resp. 76%) area on average. Compared to the scrambling architecture *SL* (resp. *SS*), the improved architecture *IL** (resp. *IS**) can also reduce 40% (resp. 22%)

[1] We add a * to indicate our proposed designs.
[2] We also did experiments for bit width $n = 6$ and 7. The results are similar to $n = 8$ and hence, are omitted due to space limit.

TABLE III
AREA OF EACH COMPONENT (μm^2).

Circuit	Area	Circuit	Area	Circuit	Area
$\sin(x)$	14.36	$\cos(x)$	13.3	e^{-x}	13.57
$\log(1+x)$	14.9	$\frac{\sin(\pi x)}{x^{0.45}}$	17.56	$\tanh(x)$	15.96
$\tanh(4x)$	10.64	$x^{2.2}$	22.61	e^{-2x}	18.89
$\frac{1}{1+e^{-x}}$	13.03		12.24	$0.5\cos(\pi x)+0.5$	8.78
LFSR(6 bit)	32.35	SSG(6 bit)	135.13	LFSR(8 bit)	43.76
SSG(8 bit)	181.68	CMP(8 bit)	26.87	DFF	4.77

area on average. Therefore, the improved architecture can significantly reduce the area over the existing ones.

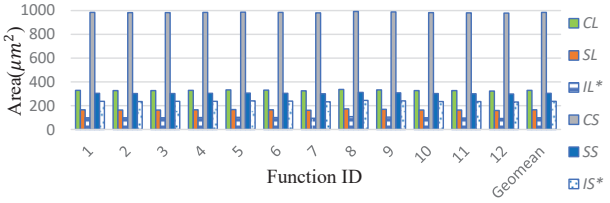


Fig. 9. Area comparison for different circuits.

We further compare the power and delay of the 3 architectures. Note that the difference among the 3 architectures lies in their randomizers as listed in Table IV. Thus, to compare their power, we only need to compare their randomizers. Clearly, the power of a DFF is smaller than that of an RNS or a CMP. Thus, the improved architecture always has a smaller power consumption than the conventional and the scrambling architectures. Furthermore, since the DFFs in the improved architecture can help reduce the critical path of the combinational part of an SC circuit [11], the delay of the improved architecture is no larger than that of the conventional and the scrambling architectures.

Together with the accuracy study, we conclude that our proposed improved univariate SC architecture has a significant hardware cost reduction compared to the existing architectures, while its accuracy is maintained at a similar level and sometimes is even higher.

TABLE IV
COMPOSITION OF THE RANDOMIZER FOR DIFFERENT ARCHITECTURES.

Architecture	Randomizer
Conventional	1 m-bit RNS, d n-bit RNSs, d CMPs
Scrambling	1 n-bit RNS, d CMPs
Improved	1 n-bit RNS, 1 CMP, $d-1$ DFFs

D. Case study: discrete cosine transform

In this section, we show an application-level case study of the improved univariate SC architecture. We apply it to discrete cosine transform (DCT). We implement $\cos(\pi x)$ by the improved univariate SC architecture and accurately implement the other calculations. After obtaining the DCT results, we apply accurate inverse DCT to recover the original image. The original image and the recovered images using the accurate $\cos(\pi x)$ are shown in Figs. 10(a) and (b), respectively. Figs. 10(c) and (d) show the recovered images using the $\cos(\pi x)$ realized by *IL** and *IS**, respectively. The peak signal-to-noise ratio (PSNR) is listed below each figure. It can be seen that the images obtained by the $\cos(\pi x)$ realized by the improved univariate SC architectures are very similar to the original image and the one obtained by the accurate $\cos(\pi x)$. Moreover, their PSNRs are around 40dB, which means only a slight degradation in quality.

(a) (b) (c) PSNR=44.0 dB (d) PSNR=38.9 dB

Fig. 10. Comparison of DCT for different methods. (a): the original image; (b): the image obtained by the accurate $\cos(\pi x)$; (c) and (d): the images obtained by the inaccurate $\cos(\pi x)$ realized by *IL** and *IS**, respectively.

VII. CONCLUSION

In this paper, we propose a low-cost high-accuracy SC architecture for univariate functions. It consists of only a single RNS, a single CMP, a minimum number of DFFs, and a few logic-free modules to improve the accuracy. We also propose a design space exploration algorithm to configure the accuracy-improvement modules. The experimental results show that the accuracy of the proposed SC architecture is close to or sometimes even higher than that of the conventional architecture, while the area is reduced by up to 76%. Our future work will extend the proposed architecture and the design space exploration method to support multivariate functions.

ACKNOWLEDGMENTS

This work is supported by the National Key R&D Program of China under Grant 2020YFB2205501.

REFERENCES

[1] S. Walczak and N. Cerpa, *Artificial Neural Networks*. New York: Academic Press, 2003, pp. 631–645.
[2] K. Parhi and Y. Liu, "Computing arithmetic functions using stochastic logic by series expansion," *IEEE TETC*, vol. 7, no. 1, pp. 44–59, 2019.
[3] A. Alaghi, W. Qian, and J. P. Hayes, "The promise and challenge of stochastic computing," *IEEE TCAD*, vol. 37, no. 8, pp. 1515–1531, 2018.
[4] T. J. Baker and J. P. Hayes, "Bayesian accuracy analysis of stochastic circuits," in *ICCAD*, 2020, pp. 1–9.
[5] Z. Zhao and W. Qian, "A general design of stochastic circuit and its synthesis," in *DATE*, 2015, pp. 1467–1472.
[6] S. Liu and J. Han, "Toward energy-efficient stochastic circuits using parallel sobol sequences," *TVLSI*, vol. 26, no. 7, pp. 1326–1339, 2018.
[7] J. H. Anderson, Y. Hara-Azumi, and S. Yamashita, "Effect of LFSR seeding, scrambling and feedback polynomial on stochastic computing accuracy," in *DATE*, 2016, pp. 1550–1555.
[8] S. A. Salehi, "Low-cost stochastic number generators for stochastic computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 4, pp. 992–1001, 2020.
[9] B. R. Gaines, "Stochastic computing," in *AFIPS Spring Joint Computer Conference*, 1967, pp. 149–156.
[10] P. Ting and J. P. Hayes, "Isolation-based decorrelation of stochastic circuits," in *ICCD*, 2016, pp. 88–95.
[11] 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.
[12] M. H. Najafi, D. J. Lilja, and M. Riedel, "Deterministic methods for stochastic computing using low-discrepancy sequences," in *ICCAD*, 2018, pp. 1–8.
[13] F. Neugebauer, I. Polian, and J. P. Hayes, "S-box-based random number generation for stochastic computing," *Microprocessors and Microsystems*, vol. 61, pp. 316–326, 2018.
[14] X. Peng and W. Qian, "Stochastic circuit synthesis by cube assignment," *IEEE TCAD*, vol. 37, no. 12, pp. 3109–3122, 2018.
[15] W. Qian *et al.*, "An architecture for fault-tolerant computation with stochastic logic," *IEEE TC*, vol. 60, no. 1, pp. 93–105, 2011.
[16] Synopsys Inc., http://www.synopsys.com.
[17] Nangate Inc., http://www.nangate.com.