A Reconfigurable Architecture with Sequential Logic-based Stochastic Computing

M. HASSAN NAJAFI, University of Minnesota
PENG LI, Intel Corporation
DAVID J. LILJA, University of Minnesota
WEIKANG QIAN, University of Michigan-Shanghai Jiao Tong University Joint Institute
KIA BAZARGAN, University of Minnesota
MARC RIEDEL, University of Minnesota

Computations based on stochastic bit streams have several advantages compared to deterministic binary radix computations, including low power consumption, low hardware cost, high fault-tolerance, and skew tolerance. To take advantage of this computing technique, previous work proposed a combinational logic-based reconfigurable architecture to perform complex arithmetic operations on stochastic streams of bits. The long execution time and the cost of converting between binary and stochastic representations, however, make the stochastic architectures less energy-efficient than the deterministic binary implementations. This paper introduces a methodology for synthesizing a given target function stochastically using finite-state machines (FSMs), and enhances and extends the reconfigurable architecture using sequential logic. Compared to the previous approach, the proposed reconfigurable architecture can save hardware area and energy consumption by up to 30% and 40%, respectively, while achieving a higher processing speed. Both stochastic reconfigurable architectures are much more tolerant of soft errors (bit flips) than the deterministic binary radix implementations, and its fault tolerance scales gracefully to very large numbers of errors.

Categories and Subject Descriptors: B.6.1 [**Logic Design**]: Design Styles–Sequential circuits; B.8.1 [**Performance and Reliability**]: Reliability, Testing and Fault-Tolerance

General Terms: Design, Performance, Reliability

Additional Key Words and Phrases: Stochastic Computing, Finite-state Machine, Polynomial Arithmetic

ACM Reference Format:
M. Hassan Najafi, Peng Li, David J. Lilja, Weikang Qian, Kia Bazargan, and Marc Riedel, 2016. A Reconfigurable Architecture with Sequential Logic-based Stochastic Computing. *ACM J. Emerg. Technol. Comput. Syst.* 0, 0, Article 0 (2017), 25 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

In stochastic computing (SC), a numeric value is represented by a bit stream consisting of zeros and ones. The positions of zeros and ones are random. The numeric value to be represented determines the probability of ones in the bit stream [Gaines 1969; Qian et al. 2011; Alaghi and Hayes 2013]. For example, "0.4" could be represented by the bit stream "00101". This bit stream has 5 bits, and the probability of each bit being one is 0.4. Note that the number of ones in the bit stream is determined by the length of the bit stream and the value it represents, but the positions of the ones are random. We can use a random number

This work was supported in part by the National Science Foundation, under grant no. CCF-1241987 and CCF-1408123, and National Natural Science Foundation of China (NSFC) under Grant No. 61472243 and 61204042. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. Portions of this work were presented in the 2012 IEEE/ACM International Conference on Computer-Aided Design [Li et al. 2012], in the 17th Asia and South Pacific Design Automation Conference [Li et al. 2012], and in the 30th IEEE International Conference on Computer Design [Li et al. 2012].
Author's addresses: M. Hassan Najafi, David J. Lilja, Kia Bazargan, and Marc Riedel, Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN, USA, 55455; Peng Li, Intel Corporation, 2111 NE 25th Ave, Hillsboro, OR, USA, 97124 (most of the research work presented in this paper was completed when the author was pursuing his Ph.D. in the Department of Electrical and Computer Engineering at University of Minnesota, Twin Cities); Weikang Qian, University of Michigan-Shanghai Jiao Tong University Joint Institute, Shanghai, China, 200240.

DOI: http://dx.doi.org/10.1145/0000000.0000000

generator, such as the one shown in Fig. 1, to convert the numeric value to the stochastic bit stream. The circuit consists of a linear feedback shift register (LFSR) and a comparator. If we want to represent x $(0 \leq x \leq 1)$ with an L-bit stochastic bit stream, we can set the LFSR to generate random numbers in the range $[0, L)$, and set the constant value to $L \cdot x$. Based on this configuration, the probability of each bit being one in the generated stochastic bit stream is x. Compared to the conventional binary radix-based deterministic computing, SC has several advantages, including low hardware cost, low power consumption, inherent fault-tolerance, and skew tolerance.

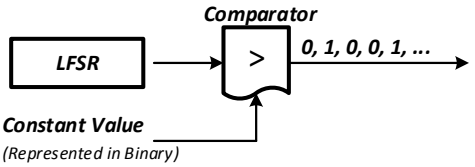

Fig. 1. Binary radix encoding to stochastic bit stream encoding converter. The comparator outputs a one if the random number generated by the linear feedback shift register (LFSR) is less than the constant value; it outputs a zero otherwise.

Stochastic computing can implement the same type of computation using much simpler logic than the conventional deterministic method. This computing technique dates back to the 1960s. In an early set of papers, researchers proposed designs that used combinational logic to implement basic arithmetic operations, such as addition and multiplication, on stochastic bit streams [Gaines 1969]. Gaines [Gaines 1969] further described ADDIE (ADaptive Digital Element), a sequential counter-based automaton for implementing arbitrary functions. In the ADDIE, however, the state of the counter is controlled in a closed-loop fashion. The problem is that ADDIE requires that the output of the counter be converted into a stochastic pulse stream in order to implement the closed-loop feedback. This requirement makes the system inefficient and requires substantial amounts of hardware.

In 2001, the implementations of some sophisticated functions using sequential logic, such as the exponentiation and hyperbolic tangent functions, were proposed [Brown and Card 2001a]. Fig. 2 shows a few examples. In Fig. 2(a), multiplication can be implemented using a single AND gate. If two input bit streams $x1$ and $x2$ are stochastically independent, we will have $P_y = P_{x1} \cdot P_{x2}$, where P_{x1}, P_{x2}, and P_y are the probability of ones in the bit streams $x1$, $x2$, and y, respectively. In Fig. 2(b), the multiply-accumulate operation, or scaled addition, can be implemented using a single multiplexer (MUX). Based on the logic function of the MUX, we can prove that $P_y = P_s \cdot P_{x1} + (1 - P_s) \cdot P_{x2}$ [Li et al. 2014a].

Fig. 2(c) shows that SC can use a single finite-state machine (FSM) to implement complex functions, such as the hyperbolic tangent and exponentiation functions, by using different state transition diagram configurations [Brown and Card 2001a]. More specifically, Fig. 3 shows the state transition diagram of the FSM for implementing the stochastic hyperbolic tangent function in Eq. 1, and Fig. 4 gives the corresponding state transition diagram for the stochastic exponentiation function in Eq. 2. In both state transition diagrams, the FSM has N states from S_0 to S_{N-1}. The input of this FSM is x, and the output is y. In fact, the only difference between these two FSMs is the output configuration. For the stochastic exponentiation function, we set $y = 1$ when the current sate is S_i, $0 \leq i < N-G$ $(0 < G \ll N)$. Otherwise, we set it to 0. For the stochastic hyperbolic tangent function, we set $y = 1$ when the current state is between S_0 and $S_{N/2}$. A detailed analysis about why the FSMs can implement the corresponding functions can be found in [Li et al. 2014b].

$$P_y = \frac{e^{N \cdot (P_x - 0.5)}}{e^{N \cdot (P_x - 0.5)} + e^{-N \cdot (P_x - 0.5)}}. \tag{1}$$

$$P_y = \begin{cases} e^{-2G(2P_x - 1)}, & 0.5 \leq P_x \leq 1, \\ 1, & 0 \leq P_x < 0.5. \end{cases} \tag{2}$$

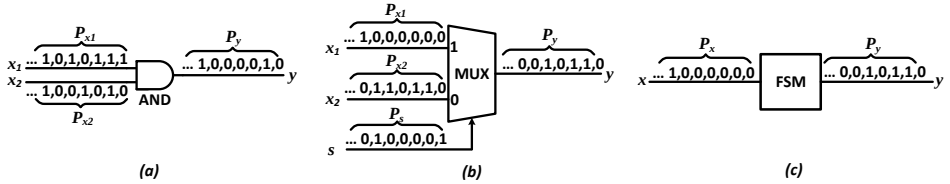

Fig. 2. Examples of SC elements. (a) Multiplication can be implemented using a single AND gate. The probability of ones in the output bit stream is the product of the probabilities of ones in the input bit streams: $P_y = P_{x1} \times P_{x2}$. (b) Multiply-accumulate operation, or scaled addition, can be implemented using a MUX: $P_y = P_s \cdot P_{x1} + (1 - P_s) \cdot P_{x2}$. (c) The hyperbolic tangent function (Eq. 1) and the exponentiation function (Eq. 2) can be implemented using a single FSM. The state transition diagram of the corresponding FSM is shown in Fig. 3 and 4.

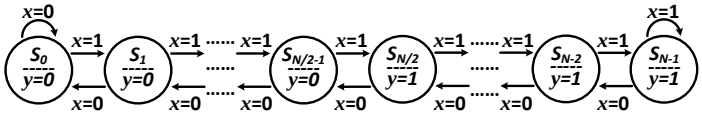

Fig. 3. State transition diagram of the FSM for implementing the hyperbolic tangent function in Eq. 1. The FSM has N states ($S_0, S_1, \cdots, S_{N-1}$). The input of this FSM is x, and the output is y [Brown and Card 2001a].

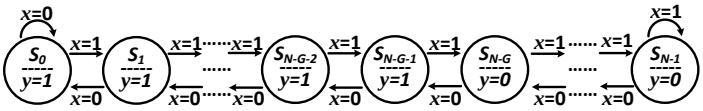

Fig. 4. State transition diagram of the FSM for implementing the exponentiation function in Eq. 2. Its state transition diagram is the same as in Fig. 3, only the output configuration differs [Brown and Card 2001a].

Stochastic computing consumes less power than conventional deterministic computing for computations such as image processing that do not require very high resolutions and can tolerate small degrees of inaccuracies [Alaghi et al. 2013]. This reduction is because its circuit structure is much simpler, and the current that is needed to drive the circuit in SC is much smaller, than what is required in the conventional deterministic implementation. Thanks to the recent advances of the rechargeable battery and solar power technology [Palacín 2009; Price et al. 2002], it is possible to power SC circuits using a small solar cell in an embedded system without recharging by an external power supply. An important point, however, is that SC systems have a higher latency than binary implementations and so consume more energy, particularly when we factor in the energy consumption of the binary-stochastic converters [Alaghi et al. 2013; Kim et al. 2016; Najafi et al. 2017; Najafi and Lilja 2017].

SC can tolerate errors due to circuit noise or bit flips. It has been shown that SC can tolerate substantially more errors than conventional deterministic computing for digital image processing applications [Li et al. 2014a] [Najafi and Salehi 2016][Qian et al. 2011]. For example, if 10% of the bits were flipped during computation, the output image of the SC circuit is still visually indistinguishable from the correct result. Future device technologies, such as nanoscale CMOS transistors, are expected to become even more sensitive to environmental noise and to process, voltage, and thermal variations [Ramanarayanan et al. 2009; Wang et al. 2011]. Thus, SC is extremely appealing for future device technologies. Recently, tolerance of variations in the arrival time of the computational units inputs has been shown to be another advantage of SC [Najafi et al. 2016]. Stochastic computing has also been applied to communications problems such as implementing low-density parity-check (LDPC) decoders [Tehrani et al. 2008], to control systems to implement proportional-integral (PI) controller [Zhang and Li 2008], and also to artificial neural networks (ANN) and deep neural networks (DNN) for hardware-efficient implementations of complicated logic functions [Brown and Card 2001b; Kim et al. 2016; Li et al. 2016; Ardakani et al. 2015; Liu et al. 2016; Li et al. 2016].

A major challenge in SC is that the functions that can be implemented using the basic logic gates are very limited. As a result, most applications in ANNs, data mining, and

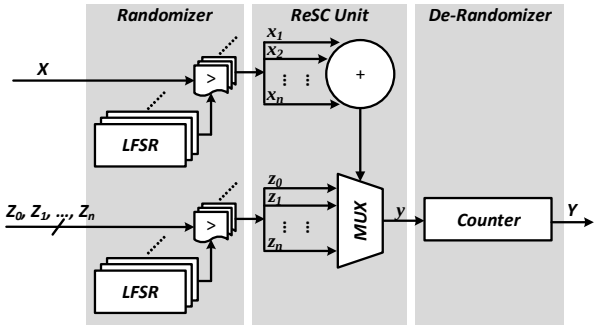

Fig. 5. The combinational logic-based reconfigurable SC architecture [Qian et al. 2011].

digital signal processing cannot benefit from this computing technique. To solve this issue, [Qian et al. 2011] proposed a combinational logic-based reconfigurable architecture, which can implement arbitrary single-input functions by changing its input parameters based on the theory of Bernstein polynomials [Lorentz 1953]. In fact, both combinational logic and sequential logic, as shown in Fig. 2, can be used to construct functions in SC. When the function is relatively simple, such as multiplication and addition, the combinational logic-based implementation requires less hardware resources than the sequential logic-based implementation. When the function is relatively complex, however, such as the exponentiation, the hyperbolic tangent, or high-order polynomial functions, the sequential logic-based implementation is more efficient since it requires fewer logic gates while producing more accurate results. Furthermore, as we will show in this work, sequential logic-based implementations can be used to implement multi-input functions at very low cost.

In our previous work, we provided a deep analysis of the existing FSM-based SC elements and implemented several digital image processing algorithms using these computing elements as case studies [Li et al. 2014b; Li et al. 2014a]. This paper makes the following contributions over our previous work: 1) we introduce new synthesis methods that can implement arbitrary target functions using different FSM structures in SC; 2) we enhance and extend the reconfigurable architecture using sequential logic to produce an architecture that consumes less hardware area and consumes less energy, while achieving a higher working frequency and the same level of fault-tolerance; 3) we study the trade-offs among the precision of the input parameter values, the hardware silicon area, the critical path latency, the power consumption, the energy dissipation and the approximation error of the reconfigurable architecture; and 4) we conduct additional experiments for fault-tolerance, including injecting soft errors in the internal architecture.

The remainder of the paper is organized as follows. Section 2 reviews the combinational logic-based reconfigurable architecture proposed by [Qian et al. 2011] and other related work in SC. Section 3 demonstrates the FSM-based reconfigurable architecture. Section 4 introduces the synthesis methodology for generating functions using different FSMs. Section 5 shows the experimental results. Conclusions are drawn in Section 6.

2. RELATED WORK

Qian *et al* [Qian et al. 2011] proposed the combinational logic-based reconfigurable architecture for performing polynomial computations on stochastic bit streams. This architecture, as shown in Fig. 5, is composed of three parts: the *Randomizer*, the *ReSC Unit*, and the *De-Randomizer*. The inputs are X and Z_0, Z_1, ..., Z_n, where n is the highest degree of a polynomial this architecture can compute, and the output is Y. These values are represented using binary radix. The architecture is reconfigurable in the sense that it can be used to compute different functions $Y = f(X)$ by setting appropriate values for the coefficients Z_i ($0 \leq i \leq n$) [Qian et al. 2011].

Their *Randomizer* uses a group of circuits shown in Fig. 1 to convert constant numerical values to stochastic bit streams. In Fig. 5, if the degree of the target polynomial is n, the *Randomizer* needs to use n pairs of LFSRs and comparators to convert X into n independent stochastic bit streams x_k ($1 \leq k \leq n$), and another $n+1$ pairs of LFSRs and comparators to convert Z_i into stochastic bit streams z_i ($0 \leq i \leq n$).

The *De-Randomizer* is implemented using a counter, which converts the resulting bit stream into a binary radix encoded value.

The *ReSC Unit*, which processes the stochastic bit streams, is the core of the architecture. It is a generalized multiplexing circuit which implements a Bernstein polynomial [Lorentz 1953] with coefficients in the unit interval. This circuit can be used to approximate arbitrary continuous functions. For example, the polynomial

$$f(X) = \frac{1}{4} + \frac{9}{8}X - \frac{15}{8}X^2 + \frac{5}{4}X^3, \tag{3}$$

can be converted into a Bernstein polynomial of degree 3:

$$f(X) = \frac{2}{8}B_{0,3}(X) + \frac{5}{8}B_{1,3}(X) + \frac{3}{8}B_{2,3}(X) + \frac{6}{8}B_{3,3}(X), \tag{4}$$

An illustration of how Eq. (4) is implemented by the *ReSC Unit* is shown in Fig. 6.

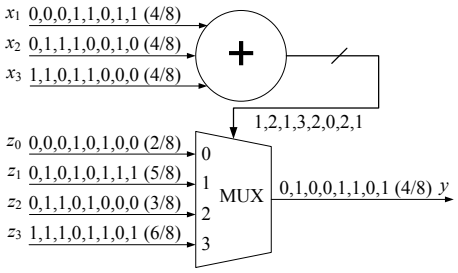

Fig. 6. *ReSC Unit* implementing the Bernstein polynomial $f(X) = \frac{2}{8}B_{0,3}(X) + \frac{5}{8}B_{1,3}(X) + \frac{3}{8}B_{2,3}(X) + \frac{6}{8}B_{3,3}(X)$ at $X = 0.5$. Independent stochastic bit streams x_1, x_2 and x_3 encode the value $X = 0.5$. Independent stochastic bit streams z_0, z_1, z_2 and z_3 encode the corresponding Bernstein coefficients.

The *ReSC Unit* consists of an adder block and a multiplexer block. The output of the adder is connected to the select bits of the multiplexer block. At every clock cycle, if the number of ones in the input set $\{x_1, \ldots, x_n\}$ equals i ($0 \leq i \leq n$), then the output of the multiplexer y is set to z_i. The output of the circuit is a stochastic bit stream y in which the probability of a bit being one equals the Bernstein polynomial $B(t) = \sum_{i=0}^{n} Z_i B_{i,n}(t)$ evaluated at $t = X$. The details of this Bernstein polynomial-based synthesis method can be found in [Qian et al. 2011]. It can be seen from Fig. 5 that the entire architecture consists of $(2n+1)$ LFSRs, $(2n+1)$ comparators, an n-bit adder, an $(n+1)$-channel multiplexer, and a counter, where n is the highest degree of the polynomial that this architecture can compute.

One of the advantages of the combinational logic-based reconfigurable architecture is that it can implement any function in the stochastic domain, as long as the function can be converted into a Bernstein polynomial. For example, the exponentiation and hyperbolic tangent functions, which were previously implemented by the FSM, can also be implemented using this architecture. However, compared to the FSM-based implementation, this architecture consumes more hardware resource. This is because in the FSM-based implementation, we only need a single LFSR to convert the input variable x into a stochastic bit stream. However, in the combinational logic-based reconfigurable architecture proposed in [Qian et al. 2011], we will need n LFSRs to convert the input variable x to n independent stochastic bit streams, where n is the degree of the Bernstein polynomial.

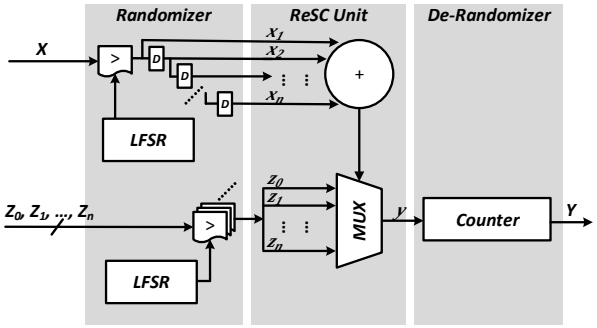

Fig. 7. The optimized ReSC architecture. Only one LFSR is used to convert the Z_i inputs into stochastic streams and $(n-1)$ LFSRs have been replaced with $(n-1)$ D-flip flops to generate the independent stochastic bit stream corresponding to input value, X. Correspondingly, the number of comparators decreases from $(2n+1)$ to $(n+2)$. Note that a further optimization could be using an LFSR and a comparator right after the MUX to generate a stochastic stream only for the selected Z_i input. This would save (n-1) comparators and so reduce the hardware cost and power consumption. However, due to a using a more complex MUX, this optimization would lead to a higher critical path and a higher area-delay product.

Recently, Ding *et al* [Ding et al. 2014] showed that the multiple constant input probabilities of the MUX-based design do not have to be independent. Based on this observation, they proposed a method to generate multiple correlated probabilities for the MUX-based SC architecture. This method can efficiently reduce the resources required to generate stochastic bit streams. However, it only works for generating constant probability values for the MUX inputs. As a result, it is not reconfigurable.

Inspired by the approach proposed in [Ding et al. 2014] and similar to the technique described in [Ichihara et al. 2014], in this paper we optimize the *Randomizer* unit of the combinational logic-based reconfigurable architecture by sharing a single LFSR, instead of using $(n+1)$ LFSRs, to generate all of the coefficient input probabilities. Since the stochastic bit streams corresponding to the Bernstein coefficient inputs of the MUX are correlation insensitive, we can use a single LFSR to convert Z_i into stochastic bit streams z_i $(0 \leq i \leq n)$. Obviously, we still need to use $(n+1)$ comparators to compare the randomly generated number with the Z_i constant inputs.

As a further optimization we use only one pair of LFSR-comparators instead of n pairs to generate the n independent stochastic bit streams corresponding to the input value, X. The single LFSR-comparator generates the first bit stream and the remaining $n-1$ streams are generated by shifting the first generated stochastic bit stream for one or a few bits using D-type flip flops. This optimization saves $(n-1)$ pairs of LFSR-comparators at the cost of a slight accuracy loss due to introducing a small amount of correlation between the X streams. By applying these architecture optimizations, the number of LFSRs required in the ReSC architecture decreases from $(2n+1)$ to only two and the number of comparators decreases from $(2n+1)$ to $(n+2)$. Thus, our optimized architecture consists of two LFSRs, $(n-1)$ D-flip flops, $(n+2)$ comparators, an n-bit adder, an $(n+1)$-channel multiplexer, and a counter. Fig. 7 shows the optimized ReSC architecture.

Saraf *et al* [Saraf et al. 2013] also proposed an FSM-based synthesis method for SC. However, that method is completely different from the method introduced by this paper. Their method first converts a target function into a corresponding Taylor series, and then constructs the FSM using the Taylor series. The approximation errors with this approach depend on two factors. One is how close the original target function is to its Taylor series. The other is how close the Taylor series is to the FSM implementation. Even if this method can obtain a perfect match between the Taylor series and the FSM implementation, the approximation error between the original target function and its Taylor series can be large. Additionally, only single-input functions and one-dimensional FSM topologies were discussed in [Saraf et al. 2013].

Although the results reported in [Saraf et al. 2013] show that their method is able to implement some single input functions using fewer states than previous methods, decreasing the number of states does not necessarily reduce the hardware resources. For example, they implement single input functions such as cos(x), sin(x), tanh(x), exp(-x), log(1+x), and x^3 using FSMs with 5 to 19 states. An FSM with N states requires $\lceil log_2 N \rceil$ flip flops. Thus, 3 to 5 flip flops are required for implementing those functions. The method that we will introduce in this paper not only is able to synthesize multi-input functions accurately, but it can also implement most functions using only an 8-state FSM. Thus, only 3-flip flops will be sufficient.

In a more recent work, [Saraf and Bazargan 2016] presents stochastic implementations of single-input polynomial functions using sequential logic. They use a stochastic integrator to generate Bernstein basis polynomials and then, similar to the ReSC architecture, they form a linear combination of the Bernstein basis polynomials with the Bernstein coefficients using a multiplexer. The inputs to the multiplexer are random bit streams representing the Bernstein coefficients, while the multiplexer select input is driven by the up/down control of the counter of the stochastic integrator. Unlike the combinational ReSC implementation that relies on multiple independent random bit streams, their work requires a single random bit-stream as the input variable. While the critical path delay is less than that of the ReSC architecture, the fault tolerance of this sequential implementation is slightly lower due to the correlation between successive counter states of the stochastic integrator. The presence of memory elements also increases the area of their sequential implementation in comparison to the ReSC architecture.

In our previous work [Li et al. 2014b], we provided in-depth analysis of existing FSM-based SC elements and explained how they work. We also implemented several digital image processing algorithms using these computing elements to demonstrate how we can utilize them in real applications [Li et al. 2014a]. In this paper, we introduce synthesis methods that can implement an arbitrary target function in the stochastic domain using FSMs. Inspired by the combinational logic-based reconfigurable architecture and the hardware saving feature of the FSM-based SC elements, we also propose a reconfigurable architecture using sequential logic, which can significantly reduce the circuit complexity for the *Randomizer*. In the following sections, we will discuss the details of the proposed architecture and the corresponding synthesis methods.

3. FSM-BASED RECONFIGURABLE ARCHITECTURE

3.1. Single-input FSM-based Reconfigurable Architecture

Inspired by the hyperbolic tangent and exponentiation functions developed by [Brown and Card 2001a], we found that these functions can be implemented using a generic reconfigurable architecture [Li et al. 2012]. In Section 1, we introduced the state transition diagram of both FSMs. Because the only difference between Figs. 3 and 4 is the output configuration of the FSM, we can implement these functions using the generic FSM-based reconfigurable architecture shown in Fig. 8. This architecture implements state machines that are similar in topology to the FSM shown in Fig. 3 and 4, but are more general in the sense that the output of each state is the state number. The architecture is composed of three parts: the *Randomizer*, the *FSM*, and the *De-Randomizer*.

The *Randomizer* has the same function as shown in Fig. 5. It converts the numerical values X and Z_i to the corresponding stochastic bit streams. However, it takes much less hardware than the combinational implementation. As we described in Section 2, the original *Randomizer* proposed in [Qian et al. 2011] uses $(2n + 1)$ pairs of LFSRs and comparators in total (n for X, $(n + 1)$ for Z_i, as shown in Fig. 5), where n is the degree of the target polynomial. The optimized *Randomizer* shown in Fig. 7 saves some hardware resources by using two LFSRs instead of $(n + 1)$, and decreasing the number of comparators to $(n + 2)$. However, it is still more expensive than the *Randomizer* shown in Fig. 8. In the proposed ar-

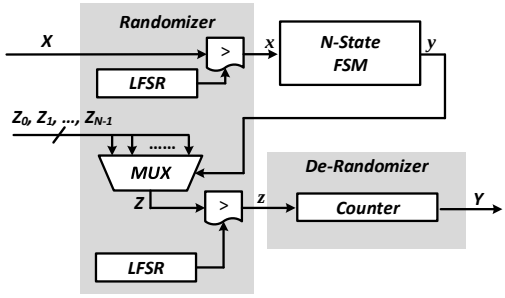

Fig. 8. An FSM-based reconfigurable stochastic architecture. The inputs are X and Z_i ($0 \leq i \leq N-1$). The output is Y. These values are represented using binary radix. The multiplexer is an M-bit N-to-1 multiplexer, where M is the number of bits of the binary radix representation of Z_i.

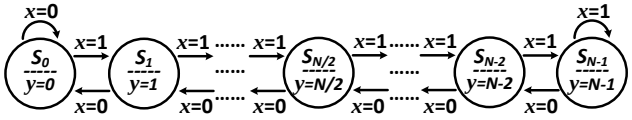

Fig. 9. The state transition diagram of an N-state Type-1 FSM. This FSM is used in the reconfigurable architecture shown in Fig. 8. The input of this FSM is x, and the numbers on each arrow represent the transition condition. This FSM has $\lceil log_2 N \rceil$ outputs, encoding a value in binary radix. In the figure, the number below each state S_i ($0 \leq i \leq N-1$) represents the output of the FSM when the current state is S_i.

chitecture, the *Randomizer* uses only two LFSRs, two comparators, and an M-bit N-channel multiplexer, where M is the number of bits of the binary radix representation of Z_i[1].

The *De-Randomizer* is implemented using a binary counter, which converts the resulting output bit stream into a binary value. This is the same as the original one shown in Fig. 5.

The state transition diagram of the FSM is shown in Fig. 9. In the remainder of the paper, we call this FSM the *Type-1 FSM*. If the current state of the Type-1 FSM is S_i, the output is $y = i$, $0 \leq i \leq N-1$. Note that if the current state of this FSM is S_i, then the *MUX* in the *Randomizer* will connect its i-th data input (i.e., Z_i) to the output of the *MUX*, and will generate the corresponding bit using the *LFSR* and the comparator. This implementation essentially has the same behavior as the circuit shown in Fig. 5, which needs $n+1$ LFSRs and $n+1$ comparators to generate $n+1$ different stochastic bit streams for the constant values $Z_0, Z_1, \cdots, Z_n$, respectively. We notice that at each clock cycle one of these random input bits to the *MUX* in the *ReSC Unit* will be selected as the output of the circuit. One way to implement this function is to choose the probability of the output bit being one using the current state number. This is equivalent to choosing the constant value in the *Randomizer* according to the current state number.

The architecture in Fig. 8 is reconfigurable in the sense that it can be used to compute different functions $Y = f(X)$ by setting appropriate values for the constants Z_i ($0 \leq i \leq N-1$). For example, if we set $Z_i = 1$ when $i \geq N/2$, and $Z_i = 0$ otherwise, it will implement the hyperbolic tangent function in Eq. 1. If we set $Z_i = 1$ when $i < N - G$, and $Z_i = 0$ otherwise, it will implement the exponentiation function in Eq. 2. In the next section, we will introduce a synthesis method that can compute the parameter values Z_i ($0 \leq i \leq N-1$) for a given target function $Y = f(X)$.

3.2. Multiple-input FSM-based Reconfigurable Architecture

Note that the architecture in Fig. 8 can only implement target functions with a single input variable. When the target functions have two input variables, we can change the architecture into a two-input FSM, as shown in Fig. 10. The state transition diagrams of the FSM can be implemented in different ways. Fig. 11 and 12 are two different state transition diagrams of a two-input FSM which can compute the target function $Y = f(X_1, X_2)$.

[1]In the combinational architecture in Fig. 5, the multiplexer is a single-bit n-to-1 multiplexer, where n is the degree of the target polynomial.

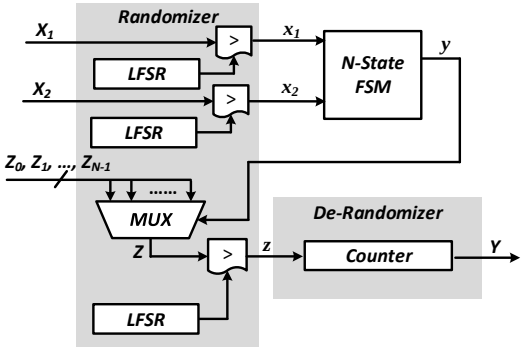

Fig. 10. The FSM-based reconfigurable stochastic architecture for implementing target functions with two input variables: $Y = f(X_1, X_2)$.

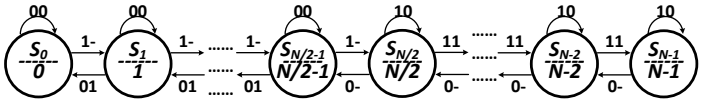

Fig. 11. The state transition diagram of an N-state Type-2A FSM. It has two inputs x_1 and x_2. The numbers on each arrow represent the transition condition, with the first corresponding to the input x_1 and the second to the input x_2. This FSM has $\lceil log_2 N \rceil$ outputs, encoding a value in binary radix. In the figure, the number below each state S_i $(0 \leq i \leq N-1)$ represents the output of the FSM when the current state is S_i.

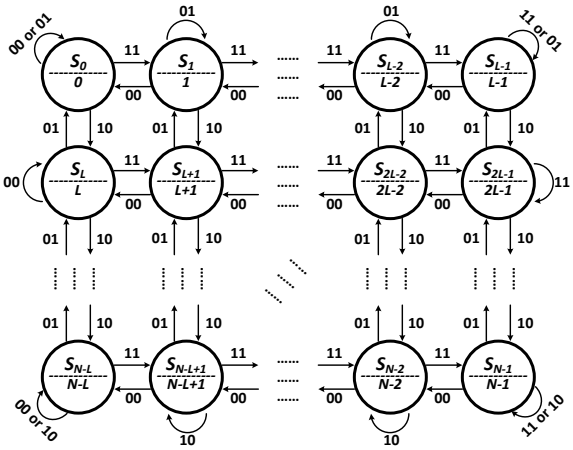

Fig. 12. The state transition diagram of an N-state Type-2B FSM which also has two inputs x_1 and x_2. The numbers on each arrow represent the transition condition, with the first corresponding to the input x_1 and the second corresponding to the input x_2. This FSM has $\lceil log_2 N \rceil$ outputs, encoding a value in binary radix. In the figure, the number below each state S_i $(0 \leq i \leq N-1)$ represents the output of the FSM when the current state is S_i. Compared to the Type-2A FSM, this FSM has a two-dimensional mesh structure. Because N is normally a power of two, we set L to $2^{\lceil 0.5 \times log_2 N \rceil}$.

In the remainder of this paper, we call the FSM in Fig. 11 the Type-2A FSM, and the one in Fig. 12 the Type-2B FSM. There are trade-offs between the different state transition diagrams since, for a given function, each architecture would present a different approximation error, number of states, and hardware resource usage. We will discuss the details in the following sections.

In fact, we can generalize the architecture to implement target functions with any number of input variables. Fig. 13 shows an implementation of the target function $Y = f(X_1, X_2, ..., X_k)$ with a k-input FSM. As the number of input variables becomes larger, we will have more flexibility to design the state transition diagram of the FSM. Fig. 14 shows an example of an FSM with three inputs. Note that neither the topology nor the state tran-

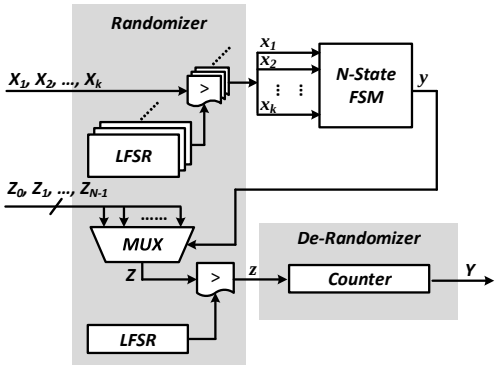

Fig. 13. The FSM-based reconfigurable stochastic architecture for implementing target functions with multiple input variables: $Y = f(X_1, X_2, ..., X_k)$.

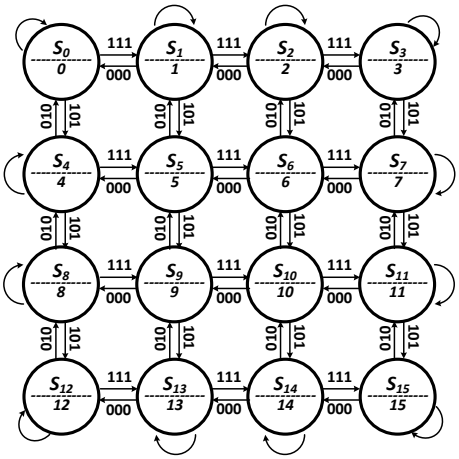

Fig. 14. The state transition diagram of a 16-state FSM which has three inputs x_1, x_2 and x_3. The numbers on each arrow represent the transition condition, with the first corresponding to the input x_1 , the second to x_2, and the third to x_3. The output of this FSM is the current state number encoded in binary radix. This FSM can be used to synthesize a three-input variable target function. Similar to the two-input FSM, the topology and state transition condition of the three-input FSM are not fixed.

sition conditions for the three-input FSM are fixed. Although this example used a mesh topology like the Type-2B FSM, we can also implement it using the linear topology like Type-1 and 2A FSM, or a cube topology. The state transition conditions for the same FSM topology can have different combinations, as long as the probability transition functions can be derived for each individual state. The details about how to derive such probability transition functions will be introduced in Section 4.

4. FSM-BASED SYNTHESIS METHOD

4.1. Synthesis Method for Single-input FSMs

To develop the FSM-based synthesis approach, we first study the state transition diagram of the Type-1 FSM. The basic form of the FSM is a set of states $S_0 \rightarrow S_{N-1}$ arranged linearly like a saturating counter. The FSM has N states in total, The input to the state machine is x and it can take on two values: '0' and '1'. The output, y, is the current state number of the FSM.

If input x is a Bernoulli random bit sequence, in which each random bit is independent and has the same probability of being a one, then the state transition process of the FSM can be modeled as a time-homogeneous Markov chain which is irreducible and ape-

riodic [Markov 1971]. Based on the theory of Markov chains, the FSM has an equilibrium state distribution.

We define the probability that each bit in the input stream x is one to be P_x, and the probability that the current state is S_i ($0 \leq i \leq N-1$) in the equilibrium (or the probability that the current output is i) to be P_i. Intuitively, P_i is a function of P_x. In the following, we derive a closed form expression for P_i in terms of P_x. This expression is used to synthesize a given target function $f(P_x)$.

Based on the theory of Markov chains [Markov 1971], in equilibrium, the probability of transitioning from the state S_{i-1} to its next state S_i equals the probability of transitioning from the state S_i to the state S_{i-1}. Thus, we have:

$$P_i \cdot (1 - P_x) = P_{i-1} \cdot P_x. \tag{5}$$

Because all of the individual state probabilities P_i must sum to unity, we have:

$$\sum_{i=0}^{N-1} P_i = 1. \tag{6}$$

Based on Eq. 5 and Eq. 6, we obtain:

$$P_i = \frac{(\frac{P_x}{1-P_x})^i}{\sum_{k=0}^{N-1} (\frac{P_x}{1-P_x})^k}. \tag{7}$$

Eq. 7 is the closed form expression for P_i in terms of P_x. In order to synthesize a target function $f(P_x)$ exactly using the FSM, we just need to find a group of weights Z_i ($0 \leq Z_i \leq 1$, and $0 \leq i \leq N-1$), so that:

$$f(P_x) = \sum_{i=0}^{N-1} Z_i \cdot P_i. \tag{8}$$

Note that we can implement this equation using the architecture in Fig. 8. The weights $(Z_0, Z_1, \cdots, Z_{N-1})$ are the data inputs to the MUX. The selection input of the MUX is the output y of the FSM. It can be seen that the data input channel is selected by the current state number at each clock cycle. In other words, the output of the MUX is $Z = Z_i$ when $y = i$. Note that z (the input of the *De-Randomizer*) is a stochastic bit stream which is converted from the output Z of the MUX. If we define the probability of ones in the bit stream z to be P_z, we have,

$$\begin{aligned} P_z &= \sum_{i=0}^{N-1} P(z = 1 \mid Z_i \; is \; selected) \cdot P(y = i) \\ &= \sum_{i=0}^{N-1} P(z_i) \cdot P_i = \sum_{i=0}^{N-1} Z_i \cdot P_i, \end{aligned} \tag{9}$$

where $P(z_i)$ is the probability of ones in the bit stream that would have been generated by the constant value Z_i directly. Note that $P(z_i) = Z_i$.

Next we show how to compute Z_i in Eq. 9 to synthesize the target function $f(P_x)$. We define the mean square error ϵ as:

$$\epsilon = \int_0^1 \left(f(P_x) - P_z\right)^2 \cdot d(P_x). \tag{10}$$

The synthesis goal is to compute Z_i to minimize ϵ [Li et al. 2012]. By expanding Eq. 10, we can rewrite ϵ as:

$$\epsilon = \int_0^1 f(P_x)^2 \cdot d(P_x) - 2 \int_0^1 f(P_x) \cdot P_z \cdot d(P_x) + \int_0^1 P_z^2 \cdot d(P_x).$$

The first term $\int_0^1 f(P_x)^2 \cdot d(P_x)$ is a constant because $f(P_x)$ is given. Thus minimizing ϵ is equivalent to minimizing the following objective function φ:

$$\varphi = \int_0^1 P_z^2 \cdot d(P_x) - 2\int_0^1 f(P_x) \cdot P_z \cdot d(P_x). \tag{11}$$

We define a vector $\mathbf{b}$, a vector $\mathbf{c}$, and a matrix $\mathbf{H}$ as follows,

$$\mathbf{b} = [Z_0, Z_1, \cdots, Z_{N-1}]^T,$$

$$\mathbf{c} = \begin{bmatrix} -\int_0^1 f(P_x) \cdot P_0 \cdot d(P_x) \\ -\int_0^1 f(P_x) \cdot P_1 \cdot d(P_x) \\ \vdots \\ -\int_0^1 f(P_x) \cdot P_{N-1} \cdot d(P_x) \end{bmatrix},$$

$$\mathbf{H} = [\mathbf{H_0}, \mathbf{H_1}, \cdots, \mathbf{H_{N-1}}]^T,$$

where P_i $(0 \leq i \leq N-1)$ in vector $\mathbf{c}$ is defined by Eq. 7 and H_i $(0 \leq i \leq N-1)$ in matrix $\mathbf{H}$ is a row vector defined as follows:

$$\mathbf{H_i} = \begin{bmatrix} \int_0^1 P_i \cdot P_0 \cdot d(P_x) \\ \int_0^1 P_i \cdot P_1 \cdot d(P_x) \\ \vdots \\ \int_0^1 P_i \cdot P_{N-1} \cdot d(P_x) \end{bmatrix}^T.$$

Referring to the expression of P_z in Eq. 9, note that:

$$\mathbf{b}^T\mathbf{H}\mathbf{b} = \int_0^1 P_z^2 \cdot d(P_x),$$

$$\mathbf{c}^T\mathbf{b} = -\int_0^1 f(P_x) \cdot P_z \cdot d(P_x),$$

Thus, the objective function φ in Eq. 11 can be rewritten as:

$$\varphi = \mathbf{b}^T\mathbf{H}\mathbf{b} + 2\mathbf{c}^T\mathbf{b}. \tag{12}$$

We notice that computing Z_i (i.e., the vector $\mathbf{b}$) to minimize φ in the form of Eq. 12 is a typical constrained quadratic programming problem. This is because P_i is a function of P_x (Eq. 7). The integral of P_i on P_x is a constant, as are the vector $\mathbf{c}$ and the matrix $\mathbf{H}$. Based on Eq. 12, the solution of Z_i (i.e., the vector $\mathbf{b}$) can be obtained using standard techniques [Golub and Van Loan 1996]. Based on this synthesis approach, if we set the target function to the hyperbolic tangent function in Eq. 1 or to the other functions, we will get exactly the same results proposed by prior work [Brown and Card 2001a; Li et al. 2014b].

4.2. Two-input FSM Analysis

The above synthesis approach also works for other FSMs, as long as their state transition processes can be modeled as time-homogeneous Markov chains [Li et al. 2012]. For example, both the Type-2A and -2B FSMs can be modeled as time-homogeneous Markov chains if their inputs are Bernoulli random bit sequences. The key here is to derive the closed form expression of P_i, which is the probability that the current state is S_i $(0 \leq i \leq N-1)$ in equilibrium.

For the Type-2A FSM, if the current state of the FSM is S_i $(0 \leq i \leq N-1)$, the next state of the FSM will be

- S_{i+1} if $x_1 = 1$ and $0 \leq i \leq \frac{N}{2} - 1$;
- S_{i-1} if $(x_1, x_2) = (0, 1)$ and $1 \leq i \leq \frac{N}{2} - 1$;

- S_{i+1} if $(x_1, x_2) = (1, 1)$ and $\frac{N}{2} \leq i \leq N-2$;
- S_{i-1} if $x_1 = 0$ and $\frac{N}{2} \leq i \leq N-1$;
- S_i in any other cases.

If the inputs x_1 and x_2 are stochastic bit streams with fixed probabilities, then the random state transition will eventually reach an equilibrium state, where the probability of transitioning from state S_i to its adjacent state S_{i+1} will equal the probability of transitioning from state S_{i+1} to state S_i. Thus, we have:

$$\begin{cases} P_i \cdot P_{x_1} = P_{i+1} \cdot (1 - P_{x_1}) \cdot P_{x_2}, & 0 \leq i \leq \frac{N}{2} - 2, \\ P_{i=\frac{N}{2}-1} \cdot P_{x_1} = P_{i=\frac{N}{2}} \cdot (1 - P_{x_1}), & \\ P_i \cdot P_{x_1} \cdot P_{x_2} = P_{i+1} \cdot (1 - P_{x_1}), & \frac{N}{2} \leq i \leq N-2. \end{cases} \tag{13}$$

where P_{x_1} and P_{x_2} are the probabilities of ones in the input x_1 and x_2, respectively. Note that state probabilities P_i must sum to unity over all S_i (similar to Eq. 6), and we have:

$$P_i = \begin{cases} \frac{(\frac{P_{x_1}}{1-P_{x_1}})^i \cdot P_{x_2}^{-i}}{\alpha}, & 0 \leq i \leq \frac{N}{2} - 1, \\ \frac{(\frac{P_{x_1}}{1-P_{x_1}})^i \cdot P_{x_2}^{i+1-N}}{\alpha}, & \frac{N}{2} \leq i \leq N-1, \end{cases} \tag{14}$$

where $\alpha = \sum_{i=0}^{\frac{N}{2}-1} (\frac{P_{x_1}}{1-P_{x_1}})^i \cdot P_{x_2}^{-i} + \sum_{i=\frac{N}{2}}^{N-1} (\frac{P_{x_1}}{1-P_{x_1}})^i \cdot P_{x_2}^{i+1-N}$.

For the Type-2B FSM, if the current state is S_i ($0 \leq i \leq N-1$), its next state will be:

- S_{i+1}, if $(x_1, x_2) = (1, 1)$ and $i \neq kL - 1$ (k is an integer, and $kL \leq N$);
- S_{i-1}, if $(x_1, x_2) = (0, 0)$ and $i \neq kL$;
- S_{i+L}, if $(x_1, x_2) = (1, 0)$ and $i < N - L$;
- S_{i-L}, if $(x_1, x_2) = (0, 1)$ and $i \geq L$.

If the inputs x_1 and x_2 are stochastic bit streams with fixed probabilities, then the random state transition will eventually reach an equilibrium state. The probability of transitioning from state S_i to its horizontal adjacent state S_{i-1} equals the probability of transitioning from the state S_{i-1} to the state S_i. Thus, we have:

$$P_i \cdot (1 - P_{x_1}) \cdot (1 - P_{x_2}) = P_{i-1} \cdot P_{x_1} \cdot P_{x_2} \tag{15}$$

In addition, the probability of transitioning from state S_i to its vertical adjacent state, S_{i-L}, equals the probability of transitioning from the state S_{i-L} to the state S_i:

$$P_i \cdot (1 - P_{x_1}) \cdot P_{x_2} = P_{i-L} \cdot P_{x_1} \cdot (1 - P_{x_2}) \tag{16}$$

Because all the individual state probabilities P_i must add up to unity, we have:

$$P_i = \frac{a^t \cdot b^s}{\sum_{u=0}^{L-1} \sum_{v=0}^{\frac{N}{L}-1} a^u \cdot b^v}, \tag{17}$$

where $s = \lfloor \frac{i}{L} \rfloor$ and $t = i$ modulo L, (i.e., $i = s \cdot L + t$), and a and b are:

$$a = \frac{P_{x_1}}{1 - P_{x_1}} \cdot \frac{P_{x_2}}{1 - P_{x_2}}, \qquad b = \frac{P_{x_1}}{1 - P_{x_1}} \cdot \frac{1 - P_{x_2}}{P_{x_2}}.$$

Equations 14 and 17 are the closed form expressions for P_i for the Type-2A and -2B FSMs, respectively. Based on these expressions, we can convert the synthesis problem into a typical constrained quadratic programming problem using a similar approach introduced in

Section 4.1. Then, we can compute the weights Z_i in Fig. 10 to implement the target function.

Note that we can also use the two-input FSMs for synthesizing target functions with one input variable. In fact, for certain single-input functions, it is not possible[2] to synthesize them using the Type-1 FSM. To synthesize single input functions with the two-input FSM, we just need to set one of the inputs as the weight. For example, if the FSM has two inputs x_1 and x_2, and the target function has a single input variable x, we can set $x_1 = x$, and find the best constant probability to use for x_2. As a result, the synthesis goal would be to compute Z_i and P_{x_2} to minimize the mean square error ϵ, which has been defined in Eq. 10. Note that if P_{x_2} is set to a constant, Z_i can be obtained using the same synthesis method introduced in Section 4.1. To compute P_{x_2}, we use an iterative approach. More specifically, we first set P_{x_2} to 0.001, and compute the corresponding Z_i and ϵ. Next, we set P_{x_2} to 0.002, and compute the corresponding Z_i and ϵ, and so on and so forth. Finally, we set P_{x_2} to 1, and compute the corresponding Z_i and ϵ. The granularity of P_{x_2} is determined by how many bits we are going to use to represent a value stochastically. In most applications, we use 1024 bits. Thus, the granularity is set to $\frac{1}{1024}$. Among these 1000 results of ϵ, we select the minimum one, and the corresponding P_{x_2} and Z_i.

4.3. Synthesis of Multi-input FSMs

The above synthesis method can also be extended to other multiple-input FSMs. The key here is to construct the state transition diagram so that we can derive the closed form expression of P_i by using the time-homogeneous Markov chain model when the inputs are stochastic bit streams. For example, we can write the following equations for the state transition diagram in Fig. 14, which is a three-input FSM:

$$P_i \cdot (1 - P_{x_1}) \cdot (1 - P_{x_2}) \cdot (1 - P_{x_3}) = P_{i-1} \cdot P_{x_1} \cdot P_{x_2} \cdot P_{x_3}. \tag{18}$$

$$P_i \cdot (1 - P_{x_1}) \cdot P_{x_2} \cdot (1 - P_{x_3}) = P_{i-4} \cdot P_{x_1} \cdot (1 - P_{x_2}) \cdot P_{x_3}. \tag{19}$$

$$\sum_{i=0}^{15} P_i = 1. \tag{20}$$

By using the above equations, we are able to derive a closed form expression of P_i in terms of P_{x_1}, P_{x_2}, and P_{x_3}. Consider constants a and b:

$$a = \frac{P_{x_1}}{1 - P_{x_1}} \cdot \frac{P_{x_2}}{1 - P_{x_2}} \cdot \frac{P_{x_3}}{1 - P_{x_3}} = \frac{P_i}{P_{i-1}},$$
$$b = \frac{P_{x_1}}{1 - P_{x_1}} \cdot \frac{1 - P_{x_2}}{P_{x_2}} \cdot \frac{P_{x_3}}{1 - P_{x_3}} = \frac{P_i}{P_{i-4}}.$$

Based on Eq. 18, 19, and 20 we have:

$$\sum_{i=0}^{15} P_i = P_{15} + \cdots + P_0 = P_{15} \cdot (1 + \frac{1}{a} + \frac{1}{a^2} + \frac{1}{a^3} + \frac{1}{b} + \frac{1}{ab} + \frac{1}{a^2 b} + \frac{1}{a^3 b} + \cdots + \frac{1}{a^3 b^3}) = 1$$

$$P_{15} = \frac{a^3.b^3}{\sum_{u=0}^{3}\sum_{v=0}^{3} a^u \cdot b^v} \Rightarrow P_i = \frac{a^t.b^s}{\sum_{u=0}^{3}\sum_{v=0}^{3} a^u \cdot b^v} \text{ where } s = \lfloor \tfrac{i}{4} \rfloor \text{ , } t = i \text{ modulo } 4.$$

Then we can use P_i to synthesize the three-input target function $f(P_{x_1}, P_{x_2}, P_{x_3})$. Note that the expression we found for the 16-state three input FSM can be easily extended to any N-state multi-input FSM as long as a similar mesh structure is used to construct the FSM.

[2]The mean square error is greater than 10^{-3} even in the optimal solution.

4.4. Examples

Prior works have proposed some FSM-based SC elements, such as the stochastic hyperbolic tangent function in Eq. 1, the stochastic exponentiation function in Eq. 2, the stochastic absolute value function in Eq. 21, and the stochastic linear gain function in Eq. 22 [Brown and Card 2001a][Li et al. 2014b].

$$f(P_x) = |P_x - 0.5| + 0.5. \tag{21}$$

$$f(P_{x_1}, P_{x_2}) = \begin{cases} 0, & 0 \le P_{x_1} \le \frac{P_{x_2}}{1+P_{x_2}}, \\ \frac{1+P_{x_2}}{1-P_{x_2}} \cdot P_{x_1} - \frac{P_{x_2}}{1-P_{x_2}}, & \frac{P_{x_2}}{1+P_{x_2}} \le P_{x_1} \le \frac{1}{1+P_{x_2}}, \\ 1, & \frac{1}{1+P_{x_2}} \le P_{x_1} \le 1. \end{cases} \tag{22}$$

All these computing elements can be implemented using the proposed FSM-based reconfigurable architecture and our synthesis method. We can use the single-input Type-1 FSM-based reconfigurable architecture to implement the stochastic exponentiation function, the stochastic hyperbolic tangent function, and the stochastic absolute value function. Because the stochastic linear gain function has two inputs, we implement this function using the two-input Type-2A FSM-based reconfigurable architecture. Table I gives the values of Z_i in the corresponding architecture for implementing different target functions. In these examples, we assume the FSM has 8 states.

Table I. The values of Z_i $(0 \le i \le 7)$ for synthesizing different target functions using different FSM-based reconfigurable architecture. The stochastic hyperbolic tangent function (Eq. 1), the stochastic exponentiation function (Eq. 2 with $G = 2$), and the stochastic absolute value function (Eq. 21) are implemented using the Type-1 FSM-based architecture. The stochastic linear gain function (Eq. 22) is implemented using the Type-2A FSM-based architecture.

Eq. 1	$Z_0 = 0$	$Z_1 = 0$	$Z_2 = 0$	$Z_3 = 0$	$Z_4 = 1$	$Z_5 = 1$	$Z_6 = 1$	$Z_7 = 1$
Eq. 2	$Z_0 = 1$	$Z_1 = 1$	$Z_2 = 1$	$Z_3 = 1$	$Z_4 = 1$	$Z_5 = 1$	$Z_6 = 0$	$Z_7 = 0$
Eq. 21	$Z_0 = 1$	$Z_1 = 0$	$Z_2 = 1$	$Z_3 = 0$	$Z_4 = 0$	$Z_5 = 1$	$Z_6 = 0$	$Z_7 = 1$
Eq. 22	$Z_0 = 1$	$Z_1 = 1$	$Z_2 = 1$	$Z_3 = 1$	$Z_4 = 0$	$Z_5 = 0$	$Z_6 = 0$	$Z_7 = 0$

We can also use the two-input FSMs to synthesize single-input functions. This method is normally adopted when the target functions are hard to synthesize using the single-input FSM. For example, the target function in Eq. 23 is a Gaussian distribution function.

$$f(P_x) = \frac{1}{\delta\sqrt{2\pi}} e^{-\frac{(2P_x - 1 - \mu)^2}{2\delta^2}}, \quad (0 \le P_x \le 1). \tag{23}$$

Although it has only one input, we are not able to synthesize it using the single-input FSM. In this case, we can use the two-input FSM. If we assume in Eq. 23 that $\delta = 2$ and $\mu = 0$, Table II gives the synthesis results when using an 8-state Type-2A FSM, and Table III gives the synthesis results when using a 16-state Type-2B FSM. In both implementations, we set the value of P_{x_2} to a constant.

Table II. The values of P_{x_2} and Z_i for the Type-2A FSM-based reconfigurable architecture when synthesizing the Gaussian distribution function in Eq. 23 with $\delta = 2$ and $\mu = 0$. These values are computed using an 8-state Type-2A FSM.

$P_{x_2} = 0.563$							
$Z_0 = 0.11$	$Z_1 = 0.64$	$Z_2 = 0.98$	$Z_3 = 0.87$	$Z_4 = 0.87$	$Z_5 = 0.98$	$Z_6 = 0.64$	$Z_7 = 0.11$

Table III. The values of P_{x_2} and Z_i for the Type-2B FSM-based reconfigurable architecture when synthesizing the Gaussian distribution function in Eq. 23 with $\delta = 2$ and $\mu = 0$. These values are computed using a 16-state Type-2B FSM.

$P_{x_2} = 0.4$							
$Z_0 = 0.11$	$Z_1 = 0$	$Z_2 = 1$	$Z_3 = 0.66$	$Z_4 = 0.8$	$Z_5 = 1$	$Z_6 = 1$	$Z_7 = 1$
$Z_8 = 1$	$Z_9 = 1$	$Z_{10} = 1$	$Z_{11} = 0$	$Z_{12} = 1$	$Z_{13} = 1$	$Z_{14} = 0.8$	$Z_{15} = 0.11$

We noticed that the values of Z_i listed in Table II and III contain not only 0 and 1 but also fractional values. If we store these values with high precision, such as an 8-bit register, they will consume a significant amount of hardware resources in the multiplexer part of the architecture. On the other hand, if we reduce the precision of these values, it may increase the approximation error. Thus, it is interesting to study the trade-offs between the approximation error and the precision of Z_i. In Fig. 15, we use simulations to compare the approximation error between the target function and the corresponding FSM-based implementations using different precisions of Z_i. Table IV summarizes the results. It can be seen that we can get an acceptable result (mean square error of less than 10^{-3}) even by storing Z_i with three bits of precision. If the application itself can tolerate some errors, it is possible to store Z_i using only a single bit.

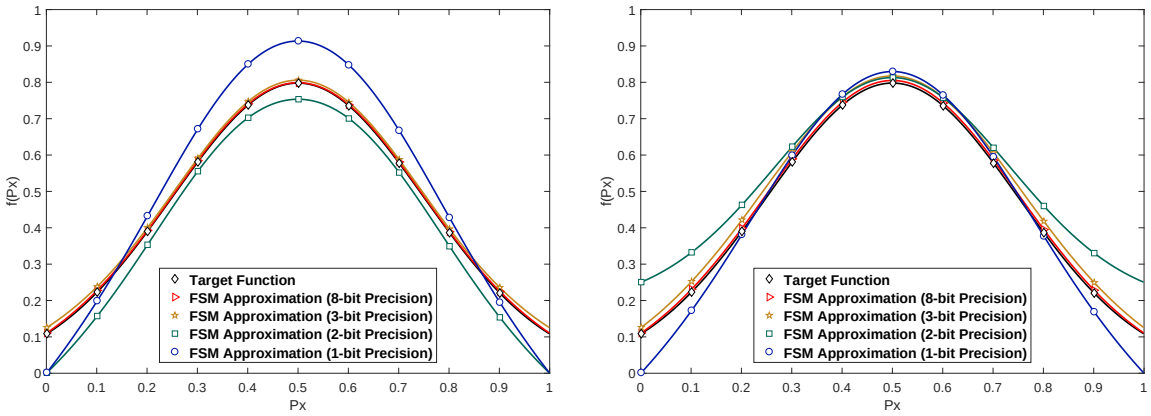

Fig. 15. (Left) In this simulation, we use four different precisions of Z_i in Table II to compare the approximation error with the target Gaussian distribution function (Eq. 23). The FSM is implemented using 8 states. (Right) In this simulation, we use four different precisions of Z_i in Table III to compare the approximation error with the target Gaussian distribution function (Eq. 23). The FSM is implemented using 16 states.

Table IV. The mean square error of the FSM-based architecture for implementing the Gaussian distribution target function (Eq. 23) when using different precisions for Z_i. The mean square error is computed using Eq. 10.

FSM	Precision of Z_i	Mean Square Error
8-State (Table II)	8-bit	1.17×10^{-5}
	3-bit	1.41×10^{-4}
	2-bit	2.70×10^{-3}
	1-bit	6.80×10^{-3}
16-State (Table III)	8-bit	5.78×10^{-5}
	3-bit	7.02×10^{-4}
	2-bit	5.80×10^{-3}
	1-bit	1.80×10^{-3}

Another example showing the versatility of our FSM-based method is the high order polynomial in Eq. 24, which was used in low-density parity-check decoding [Richardson et al. 2001]. This function cannot be synthesized using the single-input FSM. However, if we use the two-input FSM, we can synthesize this function using either of the state transition diagrams with the corresponding values shown in Tables V and VI.

$$f(P_x) = 0.1575P_x + 0.3429P_x^2 + 0.0363P_x^5 + 0.059P_x^6 + 0.279P_x^8 + 0.1253P_x^9, \quad (0 \leq P_x \leq 1). \tag{24}$$

Table V. The values of P_{x_2} and Z_i for the Type-2A FSM-based reconfigurable architecture when synthesizing the high order polynomial function in Eq. 24. These values are computed using an 8-state Type-2A FSM.

$P_{x_2} = 0.188$							
$Z_0 = 0$	$Z_1 = 0.036$	$Z_2 = 0.019$	$Z_3 = 0.04$	$Z_4 = 0.204$	$Z_5 = 0.811$	$Z_6 = 0.113$	$Z_7 = 0.999$

Table VI. The values of P_{x_2} and Z_i for the Type-2B FSM-based reconfigurable architecture when synthesizing the high order polynomial function in Eq. 24. These values are computed using a 16-state Type-2B FSM.

$P_{x_2} = 0.828$							
$Z_0 = 0$	$Z_1 = 0$	$Z_2 = 0$	$Z_3 = 0.018$	$Z_4 = 1$	$Z_5 = 0.169$	$Z_6 = 1$	$Z_7 = 0.907$
$Z_8 = 0$	$Z_9 = 1$	$Z_{10} = 0$	$Z_{11} = 0$	$Z_{12} = 1$	$Z_{13} = 1$	$Z_{14} = 1$	$Z_{15} = 1$

We also study the trade-offs between the approximation error and the precision of Z_i in this example. The results are shown in Fig. 16. We used simulations to compare the approximation error between the target function and the corresponding FSM-based implementations using different precisions of Z_i. Table VII summarizes the results. It can be seen that if we use the Type-2A FSM, we can get an acceptable result with a mean square error of less than 10^{-3} by storing Z_i with only two bits. If we use the Type-2B FSM, the mean square error is less than 10^{-4} even if Z_i is stored using only a single bit.

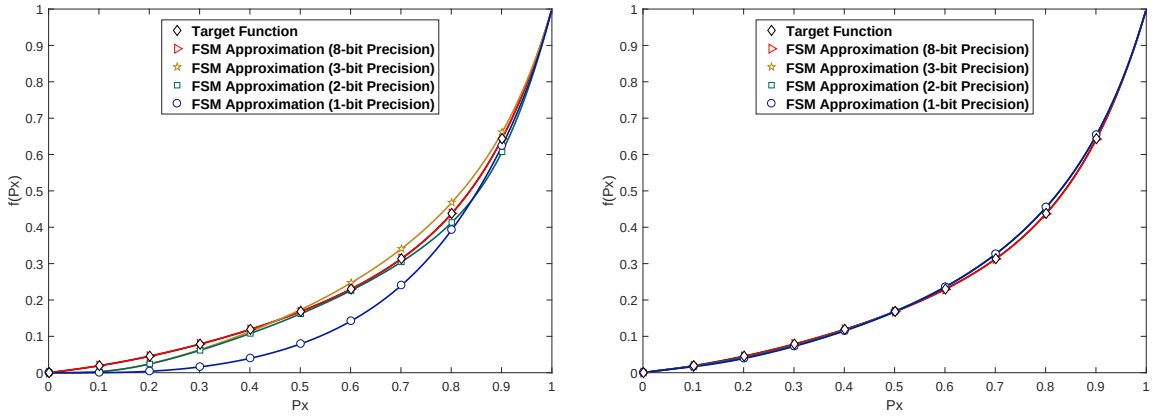

Fig. 16. (Left) In this simulation, we use four different precisions of Z_i in Table V to compare the approximation error with the target polynomial function (Eq. 24). The FSM is implemented using 8 states. (Right) In this simulation, we use four different precisions of Z_i in Table VI to compare the approximation error with the target polynomial function (Eq. 24). The FSM is implemented using 16 states.

Table VII. The mean square error of the FSM-based architecture for implementing the polynomial target function (Eq. 24) when using different precisions for Z_i. The mean square error is computed using Eq. 10.

FSM	Precision of Z_i	Mean Square Error
8-State (Table V)	8-bit	1.13×10^{-7}
	3-bit	3.25×10^{-4}
	2-bit	3.20×10^{-4}
	1-bit	3.60×10^{-3}
16-State (Table VI)	8-bit	4.72×10^{-7}
	3-bit	6.70×10^{-5}
	2-bit	6.56×10^{-5}
	1-bit	6.95×10^{-5}

In the next section, we will show the hardware area, the critical path latency, the power consumption, and the energy consumption of the FSM-based reconfigurable architecture using different precisions of Z_i.

5. EXPERIMENTAL RESULTS

In this section, we will first present a few examples on the proposed synthesis method, and then discuss the trade-offs among the precision of Z_i, the approximation error, and the hardware area. We further compare the hardware area, energy consumption, and fault-tolerance of the proposed FSM-based reconfigurable architecture with the combinational logic-based ReSC architecture [Qian et al. 2011] and the optimized ReSC architecture presented in Fig. 7.

5.1. Hardware Area and Energy Consumption Comparison

This section compares the hardware area, the maximum working frequency, the power consumption, and the energy dissipation of the three architectures. We implement the discussed architectures using Verilog HDL and evaluate the hardware area using the FreePDK45 CMOS standard cell library [Stine et al. 2007]. The Synopsys Design Compiler vH2013.12 was used to synthesize the designs. Table VIII shows the hardware area of the basic modules in the three reconfigurable architectures.

Table VIII. Hardware silicon area (μm^2) of the basic modules.

Single-bit multiplexer with different number of inputs

Number of inputs	4	5	6	7	8	9	16	32
Hardware area	16.43	21.59	30.97	37.07	41.76	46.92	78.37	155.33

Single-bit adder with different number of inputs

Number of inputs	3	4	5	6	7	8
Hardware area	8.91	24.87	25.81	34.72	35.66	59.6

10-bit LFSR

Hardware area	194.29

Comparator with different precisions

Number of bits	1	2	3	6	8	10
Hardware area	3.28	14.07	21.11	48.80	71.33	89.16

Single-bit FSM (Fig. 9)

Number of states	8	16	32
Hardware area	75.55	114.50	168.47

Two-bit FSM (Fig. 11)

Number of states	8	16	32
Hardware area	114.50	172.70	277.35

Two-bit FSM (Fig. 12)

Number of states	8	16	32
Hardware area	81.18	109.34	150.64

As discussed in Section 3, the reconfigurable architectures have several basic modules in common, such as the LFSR and the comparator. In our experiments, we use 1024 (i.e., 2^{10}) bits to represent a numerical value stochastically, so the bit width of these two components is 10. The hardware area of the original combinational logic-based reconfigurable ReSC and the optimized ReSC depends on the degree of the polynomial. For a polynomial of degree of n, the original ReSC needs $(2n+1)$ LFSRs, $(2n+1)$ comparators, an n-input adder, and an $(n+1)$-channel multiplexer, while the optimized ReSC needs two LFSRs, $(n+2)$ comparators, $(n-1)$ D-flip flops, an n-input adder, and an $(n+1)$-channel multiplexer. Table IX shows the hardware area of these architectures with different polynomial degrees (n = 3, 4, $\cdots$, 8). Clearly, for all polynomial degrees, the hardware cost of the optimized ReSC is much less than the cost of the original ReSC.

Note that we can also reduce the precision of the parameters of the optimized ReSC to reduce its hardware area. However, the area saved by this approach is very limited because

Table IX. Hardware area (μm^2) of the combinational logic-based ReSC reconfigurable architecture and the FSM-based reconfigurable architecture.

Degree n	3	4	5	6	7	8
Original ReSC (10-bit Z_i)	2036.3	2543.1	3044.3	3579.4	4048.7	4547.5
Optimized ReSC (10-bit Z_i)	1106.6	1213.6	1306.5	1431.8	1578.7	1672.6
Optimized ReSC (8-bit Z_i)	993.0	1100.0	1175.6	1297.6	1381.1	1452.0
Optimized ReSC (6-bit Z_i)	923.1	1012.3	1047.9	1115.1	1245.5	1269.5

	1-bit Z_i		2-bit Z_i		3-bit Z_i		8-bit Z_i	
	8-state	16-state	8-state	16-state	8-state	16-state	8-state	16-state
Type-1 FSM	636.8	749.9	709.1	864.9	787.5	960.2	1109.0	1485.3
Type-2A FSM	801.1	888.4	845.2	1045.6	936.7	1140.4	1248.8	1636.9
Type-2B FSM	739.6	840.5	820.8	966.8	899.2	1083.1	1205.6	1570.3

the hardware area of the MUX and the LFSR, which is used in generating the n independent X bit streams, are independent of the precision. As shown in Fig. 7, the MUX has $(n+1)$ channels. The bit width of each channel of the MUX is 1 independent of the precision the parameter Z_i. Additionally, in our experiments we set the stochastic bit stream length to 1024, and it will need at least a 10-bit LFSR to generate the input X bit stream without correlation. The hardware area of the LFSR and the comparators which are used in converting Z_i can only be reduced if we use a lower precision of Z_i. Our simulations showed that we require at least 6-bit precision of Z_i to synthesize most target functions with a mean square error of less than 10^{-3} with the ReSC architecture. The hardware silicon area of the optimized ReSC with 10, 8, and 6-bit precision for Z_i are also presented in Table IX.

The hardware area of the proposed FSM-based reconfigurable architecture is independent of the degree of the target polynomial. It depends on the type of the FSM, the number of states in the FSM, and the precision of Z_i. Note that the FSM-based architecture requires fewer LFSRs and comparators than the original ReSC. It also requires no D-flip flip and requires fewer comparators compared to the optimized ReSC. The single-input FSM-based architecture requires only two LFSRs and comparators. The two-input FSM-based configurations require three LFSRs and comparators. Note that, in the two-input FSMs, the LFSR which is used to generate the stochastic stream corresponding to the second input, $X2$, can be saved by sharing and circular shifting of the output of the LFSR which is used in converting the first input, $X1$. The circular shifting can reduce the correlation between the generated streams with no extra hardware overhead [Ichihara et al. 2014]

Table IX shows the hardware area of different configurations of the FSM-based reconfigurable architecture. It can be seen that when the number of states and the precision of Z_i are the same, the single-input Type-1 FSM requires the least silicon area, and the Type-2B FSM takes less silicon area than the Type-2A FSM. We found that most target functions can be synthesized with a mean square error of less than 10^{-3} using the 8-state Type-2B FSM with 3-bit precision for Z_i. The hardware area of this configuration is even smaller than the smallest area of the optimized combinational logic-based ReSC architecture for computing polynomials with a degree of 3, and it can save up to 30% of the hardware area for polynomials with a degree of 8. When the complexity of the target function increases, the FSM-based solution can offer significant hardware area benefits. If the applications themselves can tolerate a larger margin of error, we can further reduce the hardware area by using a lower precision of Z_i and the single-input FSM.

For both the combinational logic-based architecture and the proposed FSM-based architecture, the throughputs, which is the number of bits that can be processed per clock cycle, are exactly the same. Both architectures generate a single bit per clock cycle, and the De-Randomizer, which is simply a counter, will wait for 2^Lclock cycles to generate the final results, where L is the number of bits that represent a deterministic value. The total latency for processing each input is determined by multiplying the number of clock cycles by the latency of the critical path (the path in the entire design with the maximum delay) in each architecture. Since we process each input value for 1024 cycles, the latency of one clock

Table X. Critical path delay (ns) of the ReSC (original and optimized) and the proposed FSM-based reconfigurable architectures.

Degree n	3	4	5	6	7	8
Original ReSC (10-bit Z_i)	0.637	0.664	0.750	0.801	0.832	0.860
Optimized ReSC (10-bit Z_i)	0.600	0.656	0.701	0.730	0.749	0.772
Optimized ReSC (8-bit Z_i)	0.600	0640	0.700	0.721	0.749	0.775
Optimized ReSC (6-bit Z_i)	0.626	0.651	0.678	0.730	0.753	0.768

	1-bit Z_i		2-bit Z_i		3-bit Z_i		8-bit Z_i	
	8-state	16-state	8-state	16-state	8-state	16-state	8-state	16-state
Type-1 FSM	0.446	0.475	0.470	0.538	0.560	0.581	0.610	0.660
Type-2A FSM	0.475	0.497	0.511	0.530	0.548	0.592	0.615	0.715
Type-2B FSM	0.448	0.482	0.490	0.554	0.559	0.589	0.619	0.669

Table XI. Total Power consumption (mW) (dynamic + static) at the maximum working frequency of the ReSC (original and optimized) and the proposed FSM-based reconfigurable architectures.

Degree n	3	4	5	6	7	8
Original ReSC (10-bit Z_i)	4.987	6.048	6.041	7.037	7.392	8.722
Optimized ReSC (10-bit Z_i)	2.295	2.259	2.361	2.300	2.564	2.420
Optimized ReSC (8-bit Z_i)	2.119	2.097	2.161	2.095	2.30	2.373
Optimized ReSC (6-bit Z_i)	1.682	1.680	1.829	1.713	1.870	1.923

	1-bit Z_i		2-bit Z_i		3-bit Z_i		8-bit Z_i	
	8-state	16-state	8-state	16-state	8-state	16-state	8-state	16-state
Type-1 FSM	2.051	1.982	2.025	1.845	1.792	1.787	2.153	2.072
Type-2A FSM	2.074	2.087	1.948	1.976	1.936	1.910	2.187	2.027
Type-2B FSM	2.120	2.093	1.970	1.817	1.845	1.804	2.148	2.053

Table XII. Energy dissipation (pJ) during one clock cycle at the maximum working frequency of the ReSC (original and optimized) and the proposed FSM-based reconfigurable architectures.

Degree n	3	4	5	6	7	8
Original ReSC (10-bit Z_i)	3.176	4.015	4.530	5.636	6.150	7.500
Optimized ReSC (10-bit Z_i)	1.377	1.481	1.655	1.679	1.920	1.868
Optimized ReSC (8-bit Z_i)	1.271	1.342	1.512	1.510	1.737	1.839
Optimized ReSC (6-bit Z_i)	1.052	1.093	1.240	1.250	1.408	1.476

	1-bit Z_i		2-bit Z_i		3-bit Z_i		8-bit Z_i	
	8-state	16-state	8-state	16-state	8-state	16-state	8-state	16-state
Type-1 FSM	0.914	0.941	0.951	0.992	1.003	1.038	1.313	1.367
Type-2A FSM	0.985	1.037	0.995	1.047	1.060	1.130	1.345	1.449
Type-2B FSM	0.949	1.008	0.965	1.006	1.031	1.062	1.329	1.373

cycle, or the delay of the critical path, determines the processing speed. Table X shows the critical path delay for different configurations of the ReSC and the proposed FSM-based reconfigurable architectures. As can be seen in this table, all FSM-based architectures with low precision of Z_i (1, 2, and 3-bits) have a lower critical path latency than the optimized ReSC with 6-bit precision of Z_i. This means that for the same accuracy level, that is, an MSE less than 10^{-3}, the proposed FSM-based reconfigurable architectures is faster than the ReSC architectures.

We can evaluate the energy consumption using the product of the processing time and the total (dynamic plus static) power consumption. All of the power values can be extracted from the synthesis results. The static power or leakage is directly proportional to the hardware area and so an architecture with a lower hardware cost will have lower leakage power. Dynamic power, on the other hand, is an increasing function of the working frequency. Since the maximum working frequency is inversely proportional to the critical path latency, the architectures have different maximum working frequencies and so different processing

time. Thus, the total energy consumption ($power \times time$) is a better metric than the power consumption to compare the cost of the different architectures.

The total power consumption and the energy consumption at the maximum working frequency of each architecture are shown in Table XI and XII, respectively. As can be seen in these tables, the total power consumption of the proposed FSM-based reconfigurable architectures are of the same order or, in a few cases, even more than the power consumption of the optimized ReSC architecture with the minimum precision of Z_i. However, due to a lower critical path latency, and thus a shorter processing time, up to a 40% saving in the total energy consumption is observed when using the FSM-based architectures with 1 to 3-bit precisions instead of the low precision optimized ReSC architectures.

5.2. Fault-Tolerance Comparison

We compare the fault-tolerance capabilities of the stochastic computations on polynomial functions when the input data and the internal circuit are corrupted with noise.

5.2.1. Noise in the Input Data. We simulate noise in the input data by independently flipping the X input bits and the Z_i coefficient inputs for a given percentage of the computing elements. For example, five percent noise in the circuit means that five percent of the total number of input bits are randomly chosen and flipped. Suppose that the input data of a deterministic implementation is $M = 10$ bits. In order to have the same resolution, the bit stream of a stochastic implementation contains $2^M = 1024$ bits. We choose the error ratio λ of the input data to be 0, 0.001, 0.002, 0.005, 0.01, 0.02, 0.05 and 0.1, as measured by the fraction of random bit flips that occur [Qian and Riedel 2008][Qian et al. 2011].

To measure the impact of this noise, we performed two sets of experiments. In the first set, we chose the 6th order Maclaurin polynomial approximation of eleven elementary functions as the implementation target. We list these eleven functions in Table XIII together with the degree of the Maclaurin polynomials. Such Maclaurin approximations are commonly used in numerical evaluations of non-polynomial functions. In the second experiment, we randomly chose 100 polynomials of degree six. The goal of these experiments is to demonstrate that the proposed architectures can synthesize a large range of functions with good fault-tolerance capability.

In the first set of experiments, all of the Maclaurin polynomials evaluate to non-negative values for $0 \leq t \leq 1$. However, for some of these, the maximal evaluation on $[0, 1]$ is greater than 1. Thus, we scale these polynomials by the reciprocal of their maximum value. The scaling factors that we used are listed in Table XIII.

Table XIII. The 6th order Maclaurin polynomial approximations used as the target functions .

function	degree of polynomial	scaling factor	function	degree of polynomial	scaling factor
sin(x)	5	1	tan(x)	5	0.6818
arcsin(x)	5	0.8054	arctan(x)	5	1
sinh(x)	5	0.8511	tanh(x)	5	1
arcsinh(x)	5	1	cos(x)	6	1
cosh(x)	6	0.6481	exp(x)	6	0.3679
ln(x+1)	6	1			

We evaluated each Maclaurin polynomial on 30 points starting from 1 with an interval of $\frac{1}{32}$: 1, $\frac{31}{32}$, $\frac{30}{32}$, ..., $\frac{3}{32}$. For each error ratio λ, each Maclaurin polynomial, and each evaluation point, we simulated both the stochastic and the deterministic implementations 1000 times. We averaged the relative errors over all simulations. Finally, for each error ratio λ, we averaged the relative errors over all polynomials and all evaluation points.

In the second set of experiments, we randomly chose 100 polynomials of degree six. We evaluated each on the same 30 points. We performed similar statistics to that in the first set of experiments. Table XIV shows the average relative error of the stochastic implementation and the deterministic implementation versus different error ratios, λ, for both sets of experiments.

Table XIV. Relative error for the stochastic and deterministic implementations of polynomial computation versus the error ratio λ in the input data. All three FSMs are implemented using eight states. We set the highest degree to 6 for the combinational logic-based reconfigurable architecture (ReSC).

Error ratio λ	Maclaurin polynomial				
	Deterministic Implementation	ReSC [Qian et al. 2011]	Type-1 FSM (Fig. 9)	Type-2A FSM (Fig. 11)	Type-2B FSM (Fig. 12)
	Relative error (%)				
0.0	0.00	2.86	0.66	0.44	0.63
0.001	0.88	2.87	0.79	0.55	0.81
0.002	1.70	2.93	0.98	0.71	0.99
0.005	3.91	3.26	1.61	1.32	1.68
0.01	7.67	4.21	2.73	2.38	2.75
0.02	14.6	6.71	4.95	4.49	4.97
0.05	32.7	14.9	11.6	10.6	11.6
0.1	58.6	27.9	21.9	20.3	21.8
Error ratio λ	Randomly chosen polynomial				
	Deterministic Implementation	ReSC [Qian et al. 2011]	Type-1 FSM (Fig. 9)	Type-2A FSM (Fig. 11)	Type-2B FSM (Fig. 12)
	Relative error (%)				
0.0	0.00	3.67	3.88	0.66	0.88
0.001	1.30	3.69	3.89	0.76	0.90
0.002	2.24	3.74	3.93	0.97	1.07
0.005	4.96	4.03	3.99	1.76	1.91
0.01	9.48	4.88	4.39	3.31	3.85
0.02	18.2	7.20	5.87	6.42	7.49
0.05	42.2	15.5	11.2	15.5	18.3
0.1	79.7	29.5	21.4	29.9	35.6

When $\lambda = 0$, meaning that no noise is injected into the input data, the deterministic implementation computes without any error. However, due to the inherent variance in the stochastic input values, all of the different stochastic implementations produce a small relative error. Compared to the ReSC architecture, the Type-2A and Type2-B FSM-based architectures produce more accurate results for both the selected Malcaurin polynomials and the randomly chosen polynomials. The proposed Type-1 FSM-based architecture, on the other hand, has a higher average relative error for small λ values when processing randomly chosen polynomials. The reason is the higher approximation error when the target functions are hard to synthesize using the single-input FSM. With noise, the relative error of the deterministic implementation increases dramatically as λ increases. Even for small error rates, all the stochastic implementations perform much better, and their error tolerance capabilities are almost the same.

It is not surprising that the deterministic implementation is so sensitive to errors, given that the representation used is binary radix. In a noisy environment, bit flips afflict all the bits with equal probability. In the worst case, the most significant bit gets flipped, resulting in relative error of $2^{M-1}/2^M = 1/2$ on the input value. In contrast, in a stochastic implementation, the data is represented as the fractional weight on a bit stream of length 2^M. Thus, a single bit flip only changes the input value by $1/2^M$, which is minuscule in comparison. Furthermore, in stochastic implementations, bit flips can compensate for each other which can actually increase the accuracy of the final output.

Another phenomenon noted from Table XIV is that the relative evaluation error of randomly chosen polynomials computed with deterministic implementation is much larger than that of the Maclaurin polynomials. This is because the power-form polynomial coefficients of the randomly chosen polynomials are much larger than in the Maclaurin polynomials. In Table XIII, we listed the eleven elementary functions which are converted to Maclaurin polynomials for the deterministic implementations and the Bernstein polynomials for the combinational logic-based stochastic implementations. Their power-form polynomial coefficients are between -1 and 1. However, the power-form polynomial coefficients of the randomly chosen polynomials are much larger. We give one example below as shown in Eq. 25. Some coefficients are even larger than ten. Thus, bit flips on these coefficients could dramat-

ically change the evaluation of the randomly chosen polynomials using the deterministic implementation.

$$f(P_x) = 0.869141 - 1.74316P_x + 6.46484P_x^2 - 15.5371P_x^3 + 17.1387P_x^4 - 6.82031P_x^5 \quad (25)$$

5.2.2. Noise in the Internal Circuits. Noise on the internal circuits of the deterministic implementation is simulated by randomly flipping the bits at the inputs and output of the adder and multiplier circuits. For example, if the error ratio is 0.001, then for each bit we generate a value from a uniformly distributed probability function and compare it with the error ratio of 0.001. If the generated probability is less than the error ratio, then we will flip that bit.

For the FSM-based stochastic implementation, noise is simulated by randomly changing the current state of the FSM. For example, if the FSM has eight states and the error ratio is 0.001, then we generate a value from a uniformly distributed probability function at each clock cycle and compare it with the error ratio. If the generated probability is less than the error ratio, then we will set the current state to a random state between 0 and 7.

In all of these experiments, the bit stream of a stochastic implementation uses 1024 bits and the precision of the deterministic implementation is 10 bits. We choose the error ratio, λ, to be 0, 0.001, 0.002, 0.005, 0.01, 0.02, 0.05 and 0.1. We evaluated each Maclaurin polynomial in Table XIII on 30 points starting from 1 with an interval of $\frac{1}{32}$: 1, $\frac{31}{32}$, $\frac{30}{32}$, ..., $\frac{3}{32}$. For each error ratio λ, each Maclaurin polynomial, and each evaluation point, we simulated both the deterministic and the FSM-based stochastic implementations 1000 times. We averaged the relative errors over all simulations. Finally, for each error ratio λ, we averaged the relative errors over all polynomials and all evaluation points. Table XV shows the average relative error of the stochastic and deterministic implementations as a function of the different error ratios λ.

Table XV. Relative error of the stochastic and deterministic implementations of the Maclaurin polynomial computation versus the error ratio, λ, in the internal circuit. The FSMs are implemented using 8 states. We set the highest degree to six for the deterministic implementation.

Error ratio λ	Maclaurin polynomial			
	Deterministic Implementation	Type-1 FSM (Fig. 9)	Type-2A FSM (Fig. 11)	Type-2B FSM (Fig. 12)
	Relative error (%)			
0.001	2.52	0.83	0.64	0.78
0.002	4.42	1.07	0.89	0.99
0.005	9.81	1.90	1.71	1.67
0.01	18.5	3.28	3.08	2.77
0.02	32.6	6.01	5.69	4.93
0.05	62.9	13.0	12.3	10.8
0.1	90.1	21.8	20.7	18.6

Table XVI. Relative error for the stochastic and deterministic implementations of the Maclaurin polynomial computation versus the error ratio, λ, with simultaneous error injection on both the internal circuit and the input data. The FSMs are implemented using eight states. We set the highest degree to 6 for the deterministic implementation.

Error ratio λ	Maclaurin polynomial			
	Deterministic Implementation	Type-1 FSM (Fig. 9)	Type-2A FSM (Fig. 11)	Type-2B FSM (Fig. 12)
	Relative error (%)			
0.001	2.68	1.02	0.82	1.00
0.002	4.83	1.49	1.25	1.41
0.005	11.0	2.99	2.76	2.76
0.01	20.3	5.48	5.10	4.93
0.02	35.8	10.2	9.53	9.17
0.05	67.3	22.1	20.6	20.1
0.1	93.2	36.1	34.1	33.8

Compared to the case when the input data are corrupted with noise, the FSM-based stochastic implementation has the same level of errors for the high internal error ratios. However, compared to the deterministic implementation, it can tolerate more errors. The same argument we had for the error tolerance of stochastic bit streams when noise is injected into the input data also applies in this situation where noise is additionally injected into the internal circuitry. We also simulated injecting noise into both the input data and the internal circuits simultaneously. The results are presented in Table XVI. Clearly, the proposed FSM-based architectures are much more tolerant of noise than the deterministic binary radix-based implementation when noise affects both the input data and the internal circuit elements.

6. CONCLUSION

In this paper, we proposed and evaluated a new reconfigurable architecture for stochastic computing. The computing module of this architecture is implemented using FSMs. This architecture can compute arbitrary functions by changing its input parameters using the proposed synthesis method. Furthermore, we can make tradeoffs between hardware area and approximation error by using different configurations of the FSM, such as the number of states, the number of inputs, the state transition diagrams, and the precision of the parameters. Compared to the existing reconfigurable architecture for stochastic computing, which was implemented using combinational logic, this new architecture can reduce hardware area and energy consumption by 30% and 40%, respectively, while working at a higher speeds and delivering the same fault-tolerance capability.

REFERENCES

Armin Alaghi and John P. Hayes. 2013. Survey of Stochastic Computing. *ACM Trans. Embed. Comput. Syst.* 12, 2s, Article 92 (May 2013), 19 pages. DOI:http://dx.doi.org/10.1145/2465787.2465794

A. Alaghi, Cheng Li, and J.P. Hayes. 2013. Stochastic circuits for real-time image-processing applications. In *Design Automation Conference (DAC), 2013 50th ACM / IEEE*. 1–6.

Arash Ardakani, François Leduc-Primeau, Naoya Onizawa, Takahiro Hanyu, and Warren J. Gross. 2015. VLSI Implementation of Deep Neural Network Using Integral Stochastic Computing. *CoRR* abs/1509.08972 (2015). http://arxiv.org/abs/1509.08972

B.D. Brown and H.C. Card. 2001a. Stochastic neural computation. I. Computational elements. *Computers, IEEE Transactions on* 50, 9 (Sep 2001), 891–905. DOI:http://dx.doi.org/10.1109/12.954505

B.D. Brown and H.C. Card. 2001b. Stochastic neural computation. II. Soft competitive learning. *Computers, IEEE Transactions on* 50, 9 (Sep 2001), 906–920. DOI:http://dx.doi.org/10.1109/12.954506

Y. Ding, Y. Wu, and W. Qian. 2014. Generating multiple correlated probabilities for MUX-based stochastic computing architecture. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 519–526.

B.R. Gaines. 1969. Stochastic computing systems. In *Advances in information systems science*. Springer, New York, Philadelphia, 37–172.

G. H. Golub and C. F. Van Loan. 1996. Matrix computations (Johns Hopkins studies in mathematical sciences). (1996).

H. Ichihara, S. Ishii, D. Sunamori, T. Iwagaki, and T. Inoue. 2014. Compact and accurate stochastic circuits with shared random number sources. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. 361–366. DOI:http://dx.doi.org/10.1109/ICCD.2014.6974706

Kyounghoon Kim, Jungki Kim, Joonsang Yu, Jungwoo Seo, Jongeun Lee, and Kiyoung Choi. 2016. Dynamic Energy-accuracy Trade-off Using Stochastic Computing in Deep Neural Networks. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. ACM, New York, NY, USA, Article 124, 6 pages. DOI:http://dx.doi.org/10.1145/2897937.2898011

Bingzhe Li, M. H. Najafi, and David J. Lilja. 2016. Using Stochastic Computing to Reduce the Hardware Requirements for a Restricted Boltzmann Machine Classifier. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. ACM, New York, NY, USA, 36–41. DOI:http://dx.doi.org/10.1145/2847263.2847340

P. Li, D.J. Lilja, W. Qian, K. Bazargan, and M.D. Riedel. 2014a. Computation on Stochastic Bit Streams Digital Image Processing Case Studies. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 22, 3 (March 2014), 449–462. DOI:http://dx.doi.org/10.1109/TVLSI.2013.2247429

P. Li, D.J. Lilja, W. Qian, M.D. Riedel, and K. Bazargan. 2014b. Logical Computation on Stochastic Bit Streams with Linear Finite-State Machines. *Computers, IEEE Transactions on* 63, 6 (June 2014), 1474–1486. DOI:http://dx.doi.org/10.1109/TC.2012.231

P Li, D. J. Lilja, W. Qian, K. Bazargan, and M. D. Riedel. 2012. The synthesis of complex arithmetic computation on stochastic bit streams using sequential logic. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, 480–487.

P. Li, W. Qian, M. D. Riedel, K. Bazargan, and D. J. Lilja. 2012. The synthesis of linear finite state machine-based stochastic computational elements. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*. IEEE, 757–762.

P. Li, Weikang W. Qian, and David J Lilja. 2012. A stochastic reconfigurable architecture for fault-tolerant computation with sequential logic. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*. IEEE, 303–308.

Z. Li, A. Ren, J. Li, Q. Qiu, Y. Wang, and B. Yuan. 2016. DSCNN: Hardware-oriented optimization for Stochastic Computing based Deep Convolutional Neural Networks. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 678–681. DOI:http://dx.doi.org/10.1109/ICCD.2016.7753357

Y. Liu, H. Venkataraman, Z. Zhang, and K. K. Parhi. 2016. Machine learning classifiers using stochastic logic. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 408–411. DOI:http://dx.doi.org/10.1109/ICCD.2016.7753315

G.G. Lorentz. 1953. *Bernstein polynomials*. American Mathematical Soc.

A. Markov. 1971. Extension of the limit theorems of probability theory to a sum of variables connected in a chain. (1971).

M. H. Najafi, S. Jamali-Zavareh, D. J. Lilja, M. D. Riedel, K. Bazargan, and R. Harjani. 2017. Time-Encoded Values for Highly Efficient Stochastic Circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* PP, 99 (2017), 1–14. DOI:http://dx.doi.org/10.1109/TVLSI.2016.2645902

M. H. Najafi and D. J. Lilja. 2017. High-Speed Stochastic Circuits Using Synchronous Analog Pulses. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*.

M. H. Najafi, D. J. Lilja, M. Riedel, and K. Bazargan. 2016. Polysynchronous stochastic circuits. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. DOI:http://dx.doi.org/10.1109/ASPDAC.2016.7428060

M. H. Najafi and M. E. Salehi. 2016. A Fast Fault-Tolerant Architecture for Sauvola Local Image Thresholding Algorithm Using Stochastic Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 2 (Feb 2016), 808–812. DOI:http://dx.doi.org/10.1109/TVLSI.2015.2415932

M.R. Palacín. 2009. Recent advances in rechargeable battery materials: a chemists perspective. *Chemical Society Reviews* 38, 9 (2009), 2565–2575.

H. Price, E. Lupfert, D. Kearney, E. Zarza, G. Cohen, R. Gee, and R. Mahoney. 2002. Advances in parabolic trough solar power technology. *Journal of solar energy engineering* 124, 2 (2002), 109–125.

W. Qian, X. Li, M.D. Riedel, K. Bazargan, and D.J. Lilja. 2011. An Architecture for Fault-Tolerant Computation with Stochastic Logic. *Computers, IEEE Transactions on* 60, 1 (Jan 2011), 93–105. DOI:http://dx.doi.org/10.1109/TC.2010.202

W. Qian and M. D. Riedel. 2008. The synthesis of robust polynomial arithmetic with stochastic logic. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*. IEEE, 648–653.

R. Ramanarayanan, V. Degalahal, R. Krishnan, J. Kim, V. Narayanan, Y. Xie, M. J. Irwin, and K. Unlu. 2009. Modeling soft errors at the device and logic levels for combinational circuits. *Dependable and Secure Computing, IEEE Transactions on* 6, 3 (2009), 202–216.

T.J. Richardson, M.A. Shokrollahi, and R.L. Urbanke. 2001. Design of capacity-approaching irregular low-density parity-check codes. *Information Theory, IEEE Transactions on* 47, 2 (2001), 619–637.

Naman Saraf and Kia Bazargan. 2016. Polynomial Arithmetic Using Sequential Stochastic Logic. In *Proceedings of the 26th Edition on Great Lakes Symposium on VLSI (GLSVLSI '16)*. ACM, New York, NY, USA, 245–250. DOI:http://dx.doi.org/10.1145/2902961.2902981

N. Saraf, K. Bazargan, D. J. Lilja, and M. D. Riedel. 2013. Stochastic functions using sequential logic. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*. IEEE, 507–510.

J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and others. 2007. FreePDK: An open-source variation-aware design kit. In *Microelectronic Systems Education, 2007. MSE'07. IEEE International Conference on*. IEEE, 173–174.

S.S. Tehrani, S. Mannor, and W.J. Gross. 2008. Fully Parallel Stochastic LDPC Decoders. *Signal Processing, IEEE Transactions on* 56, 11 (Nov 2008), 5692–5703. DOI:http://dx.doi.org/10.1109/TSP.2008.929671

P. Wang, X. Wang, Y. Zhang, H. Li, S. P. Levitan, and Y. Chen. 2011. Nonpersistent errors optimization in spin-MOS logic and storage circuitry. *Magnetics, IEEE Transactions on* 47, 10 (2011), 3860–3863.

D. Zhang and H. Li. 2008. A Stochastic-Based FPGA Controller for an Induction Motor Drive With Integrated Neural Network Algorithms. *Industrial Electronics, IEEE Transactions on* 55, 2 (Feb 2008), 551–561. DOI:http://dx.doi.org/10.1109/TIE.2007.911946