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.


General Terms: Design, Performance, Reliability

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

ACM Reference Format:
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.

© 2017 ACM. 1550-4832/2017/-ART0 $15.00
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.

![Diagram of stochastic bit stream encoding converter.](image)

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 \( x_1 \) and \( x_2 \) are stochastically independent, we will have \( P_y = P_{x_1} \cdot P_{x_2} \), where \( P_{x_1} \) and \( P_{x_2} \) are the probability of ones in the bit streams \( x_1 \), \( x_2 \), 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 \( P_y = P_0 \cdot P_{x_1} + (1 - P_0) \cdot P_{x_2} \) [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 state 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].

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

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

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].

![Fig. 5. The combinational logic-based reconfigurable SC architecture [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

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

can be converted into a Bernstein polynomial of degree 3:

\[
   f(X) = \frac{2}{8}B_{0,3}(X) + \frac{5}{8}B_{1,3}(X) + \frac{3}{8}B_{2,3}(X) + \frac{6}{8}B_{3,3}(X),
\]

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

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.
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 \( S_i \).

The architecture, 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 \( Z_i \).

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 \( Z_0, Z_1, \ldots, Z_n \), 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 \( Y = f(X_1, X_2) \).

---

1In 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 \( Y = f(X_1, X_2, \ldots, X_k) \) 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: $Y = f(X_1, X_2, \ldots, X_k)$.

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.

4. FSM-BASED SYNTHESIS METHOD

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
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:

\[
P_i \cdot (1 - P_x) = P_{i-1} \cdot P_x.
\]

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

\[
\sum_{i=0}^{N-1} P_i = 1.
\]

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

\[
P_i = \frac{(\frac{P_i}{1 - P_x})^i}{\sum_{k=0}^{N-1} (\frac{P_k}{1 - P_x})^k}.
\]

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:

\[
f(P_x) = \sum_{i=0}^{N-1} Z_i \cdot P_i.
\]

Note that we can implement this equation using the architecture in Fig. 8. The weights \( (Z_0, Z_1, \ldots, Z_{N-1}) \) 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:

\[
P_z = \sum_{i=0}^{N-1} P(z = 1 \mid Z_i \text{ is selected}) \cdot P(y = i)
\]

\[
= \sum_{i=0}^{N-1} P(z_i) \cdot P_i = \sum_{i=0}^{N-1} Z_i \cdot P_i,
\]

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 \( \epsilon \) as:

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

The synthesis goal is to compute \( Z_i \) to minimize \( \epsilon \) [Li et al. 2012]. By expanding Eq. 10, we can rewrite \( \epsilon \) 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 $\int_0^1 f(P_z)^2 \cdot d(P_z)$ is a constant because $f(P_z)$ is given. Thus minimizing $\epsilon$ is equivalent to minimizing the following objective function $\varphi$:

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

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

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

$$\mathbf{c} = \left[ \begin{array}{c}
- \int_0^1 f(P_z) \cdot P_0 \cdot d(P_z) \\
- \int_0^1 f(P_z) \cdot P_1 \cdot d(P_z) \\
\vdots \\
- \int_0^1 f(P_z) \cdot P_{N-1} \cdot d(P_z)
\end{array} \right],$$

$$\mathbf{H} = [\mathbf{H}_0, \mathbf{H}_1, \cdots, \mathbf{H}_{N-1}]^T,$$

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:

$$\mathbf{H}_i = \left[ \begin{array}{c}
\int_0^1 P_i \cdot P_0 \cdot d(P_z) \\
\int_0^1 P_i \cdot P_1 \cdot d(P_z) \\
\vdots \\
\int_0^1 P_i \cdot P_{N-1} \cdot d(P_z)
\end{array} \right]^T.$$

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

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

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

Thus, the objective function $\varphi$ in Eq. 11 can be rewritten as:

$$\varphi = \mathbf{b}^T \mathbf{H} \mathbf{b} + 2\mathbf{c}^T \mathbf{b}. \hspace{1cm} (12)$$

We notice that computing $Z_i$ (i.e., the vector $\mathbf{b}$) to minimize $\varphi$ in the form of Eq. 12 is a typical constrained quadratic programming problem. This is because $P_i$ is a function of $P_z$ (Eq. 7). The integral of $P_i$ on $P_z$ 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 $1 \leq i \leq \frac{N}{2} - 1$;
In addition, 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:

$$
\begin{align*}
P_i &\cdot P_{z_1} = P_{i+1} \cdot (1 - P_{z_1}) \cdot P_{z_2}, \\
P_i &\cdot P_{z_2} = P_{i+1} \cdot (1 - P_{z_1}),
\end{align*}
$$

where $P_{z_1}$ and $P_{z_2}$ 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:

$$
P_i = \begin{cases} 
\frac{P_{z_1} - P_{z_2}}{\alpha}, & 0 \leq i \leq \lfloor \frac{N}{2} \rfloor - 1, \\
\frac{P_{z_1}^{i+1} - P_{z_2}}{\alpha}, & \lfloor \frac{N}{2} \rfloor \leq i \leq N - 1,
\end{cases}
$$

where $\alpha = \sum_{i=0}^{\lfloor \frac{N}{2} \rfloor - 1} (P_{z_1} - P_{z_2})^i \cdot P_{z_2}^{-i} + \sum_{i=\lfloor \frac{N}{2} \rfloor}^{N-1} (P_{z_1} - P_{z_2})^i \cdot P_{z_2}^{-i+1-N}$.

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, where 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:

$$
P_i \cdot (1 - P_{z_1}) \cdot (1 - P_{z_2}) = P_{i-1} \cdot P_{z_1} \cdot P_{z_2}
$$

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 $S_i$:

$$
P_i \cdot (1 - P_{z_1}) \cdot P_{z_2} = P_{i-L} \cdot P_{z_1} \cdot (1 - P_{z_2})
$$

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

$$
P_i = \frac{a^t \cdot b^s}{\sum_{u=0}^{L-1} \sum_{v=0}^{\lfloor \frac{N}{2} \rfloor - 1} a^u \cdot b^v},
$$

where $s = \left\lfloor \frac{i}{2} \right\rfloor$ and $t = i$ modulo $L$, (i.e., $i = s \cdot L + t$), and $a$ and $b$ are:

$$
a = \frac{P_{z_1}}{1 - P_{z_1}} \cdot \frac{P_{z_2}}{1 - P_{z_2}}, \quad b = \frac{P_{z_1}}{1 - P_{z_1}} \cdot \frac{1 - P_{z_2}}{P_{z_2}}.
$$

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 \( \epsilon \), 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 \( \epsilon \). Next, we set \( P_{x_2} \) to 0.002, and compute the corresponding \( Z_i \) and \( \epsilon \), and so on and so forth. Finally, we set \( P_{x_2} \) to 1, and compute the corresponding \( Z_i \) and \( \epsilon \). 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 \( \epsilon \), 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 \( P_t \) 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:

\[
P_t \cdot (1 - P_{x_1}) \cdot (1 - P_{x_2}) \cdot (1 - P_{x_3}) = P_{t-1} \cdot P_{x_1} \cdot P_{x_2} \cdot P_{x_3},
\]

\[
P_t \cdot (1 - P_{x_1}) \cdot P_{x_2} \cdot (1 - P_{x_3}) = P_{t-4} \cdot P_{x_1} \cdot (1 - P_{x_2}) \cdot P_{x_3}.
\]

\[
\sum_{i=0}^{15} P_i = 1.
\]

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 \( b \):

\[
a = \frac{P_{x_1}}{1 - P_{x_1}}, \quad b = \frac{P_{x_2}}{1 - P_{x_2}}.
\]

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

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

\[
P_{15} = \frac{a^3b^3}{\sum_{u=0}^{3} \sum_{v=0}^{3} a^u \cdot b^v} \quad \text{where} \quad s = \lfloor \frac{i}{4} \rfloor, \quad t = i \mod 4.
\]

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].

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

\[
f(P_{x_1}, P_{x_2}) = \begin{cases} 
0, & 0 \leq P_{x_1} \leq \frac{P_{x_2}}{1 + P_{x_2}}, \\
1 - \frac{P_{x_2}}{1 + P_{x_2}}, & \frac{P_{x_2}}{1 + P_{x_2}} \leq P_{x_1} \leq \frac{P_{x_2}}{1 + P_{x_2}}, \\
1, & \frac{P_{x_2}}{1 + P_{x_2}} \leq P_{x_1} \leq 1.
\end{cases} \tag{22}
\]

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.

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.

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

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 I. The values of \(Z_i\) (0 ≤ \(i\) ≤ 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.

<table>
<thead>
<tr>
<th>Eq. 1</th>
<th>(Z_0 = 0)</th>
<th>(Z_1 = 0)</th>
<th>(Z_2 = 0)</th>
<th>(Z_3 = 0)</th>
<th>(Z_4 = 1)</th>
<th>(Z_5 = 1)</th>
<th>(Z_6 = 1)</th>
<th>(Z_7 = 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eq. 2</td>
<td>(Z_0 = 1)</td>
<td>(Z_1 = 1)</td>
<td>(Z_2 = 1)</td>
<td>(Z_3 = 1)</td>
<td>(Z_4 = 1)</td>
<td>(Z_5 = 1)</td>
<td>(Z_6 = 0)</td>
<td>(Z_7 = 0)</td>
</tr>
<tr>
<td>Eq. 21</td>
<td>(Z_0 = 1)</td>
<td>(Z_1 = 1)</td>
<td>(Z_2 = 1)</td>
<td>(Z_3 = 1)</td>
<td>(Z_4 = 0)</td>
<td>(Z_5 = 1)</td>
<td>(Z_6 = 0)</td>
<td>(Z_7 = 1)</td>
</tr>
<tr>
<td>Eq. 22</td>
<td>(Z_0 = 1)</td>
<td>(Z_1 = 1)</td>
<td>(Z_2 = 1)</td>
<td>(Z_3 = 1)</td>
<td>(Z_4 = 0)</td>
<td>(Z_5 = 0)</td>
<td>(Z_6 = 0)</td>
<td>(Z_7 = 0)</td>
</tr>
</tbody>
</table>

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.

| \(P_{x_2} = 0.4\) | \(Z_0 = 0.11\) | \(Z_1 = 0.64\) | \(Z_2 = 0.98\) | \(Z_3 = 0.66\) | \(Z_4 = 0.8\) | \(Z_5 = 1\) | \(Z_6 = 0.64\) | \(Z_7 = 1\) |
| \(Z_8 = 1\) | \(Z_9 = 0\) | \(Z_{10} = 1\) | \(Z_{11} = 0\) | \(Z_{12} = 1\) | \(Z_{13} = 1\) | \(Z_{14} = 0.8\) | \(Z_{15} = 0.11\) |
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](image_url)

**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.

<table>
<thead>
<tr>
<th>FSM</th>
<th>Precision of $Z_i$</th>
<th>Mean Square Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-State</td>
<td>8-bit</td>
<td>$1.17 \times 10^{-5}$ (Table II)</td>
</tr>
<tr>
<td></td>
<td>3-bit</td>
<td>$1.41 \times 10^{-4}$</td>
</tr>
<tr>
<td></td>
<td>2-bit</td>
<td>$2.70 \times 10^{-3}$</td>
</tr>
<tr>
<td></td>
<td>1-bit</td>
<td>$6.80 \times 10^{-3}$</td>
</tr>
<tr>
<td>16-State</td>
<td>8-bit</td>
<td>$5.78 \times 10^{-5}$ (Table III)</td>
</tr>
<tr>
<td></td>
<td>3-bit</td>
<td>$7.02 \times 10^{-4}$</td>
</tr>
<tr>
<td></td>
<td>2-bit</td>
<td>$5.80 \times 10^{-3}$</td>
</tr>
<tr>
<td></td>
<td>1-bit</td>
<td>$1.80 \times 10^{-3}$</td>
</tr>
</tbody>
</table>

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.

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

(24)

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 \( 10^{-1} \) even if \( Z_i \) is stored using only a single bit.

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.

<table>
<thead>
<tr>
<th>FSM</th>
<th>Precision of ( Z_i )</th>
<th>Mean Square Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-State (Table V)</td>
<td>8-bit</td>
<td>( 1.13 \times 10^{-7} )</td>
</tr>
<tr>
<td></td>
<td>3-bit</td>
<td>( 3.25 \times 10^{-4} )</td>
</tr>
<tr>
<td></td>
<td>2-bit</td>
<td>( 3.20 \times 10^{-4} )</td>
</tr>
<tr>
<td></td>
<td>1-bit</td>
<td>( 3.60 \times 10^{-3} )</td>
</tr>
<tr>
<td>16-State (Table VI)</td>
<td>8-bit</td>
<td>( 4.72 \times 10^{-7} )</td>
</tr>
<tr>
<td></td>
<td>3-bit</td>
<td>( 6.70 \times 10^{-5} )</td>
</tr>
<tr>
<td></td>
<td>2-bit</td>
<td>( 6.56 \times 10^{-5} )</td>
</tr>
<tr>
<td></td>
<td>1-bit</td>
<td>( 6.95 \times 10^{-5} )</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Table VIII. Hardware silicon area ($\mu$m$^2$) of the basic modules.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of inputs</td>
</tr>
<tr>
<td>Hardware area</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Single-bit adder with different number of inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of inputs</td>
</tr>
<tr>
<td>Hardware area</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10-bit LFSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware area</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Comparator with different precisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of bits</td>
</tr>
<tr>
<td>Hardware area</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Single-bit FSM (Fig. 9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of states</td>
</tr>
<tr>
<td>Hardware area</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Two-bit FSM (Fig. 11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of states</td>
</tr>
<tr>
<td>Hardware area</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Two-bit FSM (Fig. 12)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of states</td>
</tr>
<tr>
<td>Hardware area</td>
</tr>
</tbody>
</table>

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...
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 flop 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, \( X \), can be saved by sharing and circular shifting of the output of the LFSR which is used in converting the first input, \( X \). 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 \( 2^L \) clock 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 cycle is.

<table>
<thead>
<tr>
<th>Degree ( n )</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original ReSC (10-bit ( Z_i ))</td>
<td>2036.3</td>
<td>2543.1</td>
<td>3044.3</td>
<td>3579.4</td>
<td>4048.7</td>
<td>4547.5</td>
</tr>
<tr>
<td>Optimized ReSC (10-bit ( Z_i ))</td>
<td>1108.6</td>
<td>1213.6</td>
<td>1306.5</td>
<td>1431.8</td>
<td>1578.7</td>
<td>1672.6</td>
</tr>
<tr>
<td>Optimized ReSC (8-bit ( Z_i ))</td>
<td>993.9</td>
<td>1175.6</td>
<td>1297.8</td>
<td>1381.1</td>
<td>1452.0</td>
<td>1578.7</td>
</tr>
<tr>
<td>Optimized ReSC (6-bit ( Z_i ))</td>
<td>923.1</td>
<td>1012.3</td>
<td>1047.9</td>
<td>1115.1</td>
<td>1245.5</td>
<td>1280.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( Z_i )</th>
<th>1-bit ( Z_i )</th>
<th>2-bit ( Z_i )</th>
<th>3-bit ( Z_i )</th>
<th>8-bit ( Z_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-state</td>
<td>16-state</td>
<td>8-state</td>
<td>16-state</td>
<td>8-state</td>
</tr>
<tr>
<td>Type-1 FSM</td>
<td>636.8</td>
<td>749.9</td>
<td>709.1</td>
<td>864.9</td>
</tr>
<tr>
<td>Type-2A FSM</td>
<td>801.1</td>
<td>888.4</td>
<td>845.2</td>
<td>1045.6</td>
</tr>
<tr>
<td>Type-2B FSM</td>
<td>739.6</td>
<td>840.5</td>
<td>820.8</td>
<td>966.8</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Degree $n$</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original ReSC (10-bit $Z_i$)</td>
<td>0.637</td>
<td>0.664</td>
<td>0.750</td>
<td>0.801</td>
<td>0.832</td>
<td>0.860</td>
</tr>
<tr>
<td>Optimized ReSC (10-bit $Z_i$)</td>
<td>0.600</td>
<td>0.656</td>
<td>0.701</td>
<td>0.730</td>
<td>0.749</td>
<td>0.772</td>
</tr>
<tr>
<td>Optimized ReSC (6-bit $Z_i$)</td>
<td>0.626</td>
<td>0.651</td>
<td>0.678</td>
<td>0.730</td>
<td>0.753</td>
<td>0.768</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Degree $n$</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original ReSC (10-bit $Z_i$)</td>
<td>4.987</td>
<td>6.048</td>
<td>6.041</td>
<td>7.037</td>
<td>7.392</td>
<td>8.722</td>
</tr>
<tr>
<td>Optimized ReSC (10-bit $Z_i$)</td>
<td>2.295</td>
<td>2.259</td>
<td>2.361</td>
<td>2.300</td>
<td>2.364</td>
<td>2.420</td>
</tr>
<tr>
<td>Optimized ReSC (8-bit $Z_i$)</td>
<td>2.119</td>
<td>2.161</td>
<td>2.124</td>
<td>2.137</td>
<td>2.131</td>
<td>2.185</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Degree $n$</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original ReSC (10-bit $Z_i$)</td>
<td>3.176</td>
<td>4.015</td>
<td>4.530</td>
<td>5.636</td>
<td>6.150</td>
<td>7.500</td>
</tr>
<tr>
<td>Optimized ReSC (10-bit $Z_i$)</td>
<td>1.377</td>
<td>1.481</td>
<td>1.655</td>
<td>1.679</td>
<td>1.920</td>
<td>1.986</td>
</tr>
<tr>
<td>Optimized ReSC (8-bit $Z_i$)</td>
<td>1.271</td>
<td>1.342</td>
<td>1.512</td>
<td>1.510</td>
<td>1.737</td>
<td>1.839</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Degree $n$</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original ReSC (10-bit $Z_i$)</td>
<td>0.914</td>
<td>0.941</td>
<td>0.951</td>
<td>0.992</td>
<td>1.003</td>
<td>1.038</td>
</tr>
<tr>
<td>Optimized ReSC (10-bit $Z_i$)</td>
<td>0.985</td>
<td>1.037</td>
<td>0.995</td>
<td>1.047</td>
<td>1.060</td>
<td>1.130</td>
</tr>
<tr>
<td>Optimized ReSC (6-bit $Z_i$)</td>
<td>0.949</td>
<td>1.008</td>
<td>0.965</td>
<td>1.006</td>
<td>1.031</td>
<td>1.062</td>
</tr>
</tbody>
</table>

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 × 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 $\lambda$ 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>
<thead>
<tr>
<th>Function</th>
<th>Degree of Polynomial</th>
<th>Scaling Factor</th>
<th>Function</th>
<th>Degree of Polynomial</th>
<th>Scaling Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin(x)</td>
<td>5</td>
<td>1</td>
<td>tan(x)</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>arcsin(x)</td>
<td>5</td>
<td>0.8054</td>
<td>arctan(x)</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>sinh(x)</td>
<td>5</td>
<td>0.8511</td>
<td>tanh(x)</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>arcsinh(x)</td>
<td>5</td>
<td>1</td>
<td>cos(x)</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>cosh(x)</td>
<td>6</td>
<td>0.6481</td>
<td>exp(x)</td>
<td>6</td>
<td>0.3679</td>
</tr>
<tr>
<td>ln(x+1)</td>
<td>6</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We evaluated each Maclaurin polynomial on 30 points starting from 1 with an interval of $\frac{1}{32} : 1, \frac{31}{32}, \frac{30}{32}, \ldots, \frac{3}{32}$. For each error ratio $\lambda$, 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 $\lambda$, 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, $\lambda$, for both sets of experiments.
Table XIV. Relative error for the stochastic and deterministic implementations of polynomial computation versus the error ratio $\lambda$ 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).

<table>
<thead>
<tr>
<th>Error ratio $\lambda$</th>
<th>Maclaurin polynomial</th>
<th>Randomly chosen polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Deterministic</td>
<td>ReSC [Qian et al. 2011]</td>
</tr>
<tr>
<td></td>
<td>Relative error (%)</td>
<td>Relative error (%)</td>
</tr>
<tr>
<td>0.0</td>
<td>0.00</td>
<td>2.86</td>
</tr>
<tr>
<td>0.001</td>
<td>0.88</td>
<td>2.87</td>
</tr>
<tr>
<td>0.002</td>
<td>1.70</td>
<td>2.93</td>
</tr>
<tr>
<td>0.005</td>
<td>3.91</td>
<td>3.26</td>
</tr>
<tr>
<td>0.01</td>
<td>7.67</td>
<td>4.21</td>
</tr>
<tr>
<td>0.02</td>
<td>14.6</td>
<td>6.71</td>
</tr>
<tr>
<td>0.05</td>
<td>32.7</td>
<td>14.9</td>
</tr>
<tr>
<td>0.1</td>
<td>58.6</td>
<td>27.9</td>
</tr>
</tbody>
</table>

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 Maclaurin 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 $\lambda$ 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 $\lambda$ 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.

\[ f(P_x) = 0.869141 - 1.74316P_x + 6.46484P_x^2 - 15.5371P_x^3 + 17.1387P_x^4 - 6.82031P_x^5 \] (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, \( \lambda \), 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 \( \frac{1}{32}, \frac{31}{32}, \frac{61}{32}, \ldots, \frac{3}{2} \). For each error ratio \( \lambda \), 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 \( \lambda \), 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 \( \lambda \).

<table>
<thead>
<tr>
<th>Error ratio ( \lambda )</th>
<th>Relative error (%)</th>
<th>Maclaurin polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Deterministic</td>
<td>Type-1</td>
</tr>
<tr>
<td></td>
<td>Implementation</td>
<td>FSM (Fig. 9)</td>
</tr>
<tr>
<td>0.001</td>
<td>2.52</td>
<td>0.83</td>
</tr>
<tr>
<td>0.002</td>
<td>4.42</td>
<td>1.07</td>
</tr>
<tr>
<td>0.005</td>
<td>9.81</td>
<td>1.90</td>
</tr>
<tr>
<td>0.01</td>
<td>18.5</td>
<td>3.28</td>
</tr>
<tr>
<td>0.02</td>
<td>32.6</td>
<td>6.01</td>
</tr>
<tr>
<td>0.05</td>
<td>62.9</td>
<td>13.0</td>
</tr>
<tr>
<td>0.1</td>
<td>90.1</td>
<td>21.8</td>
</tr>
</tbody>
</table>

Table XVI. Relative error for the stochastic and deterministic implementations of the Maclaurin polynomial computation versus the error ratio, \( \lambda \), 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.

<table>
<thead>
<tr>
<th>Error ratio ( \lambda )</th>
<th>Relative error (%)</th>
<th>Maclaurin polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Deterministic</td>
<td>Type-1</td>
</tr>
<tr>
<td></td>
<td>Implementation</td>
<td>FSM (Fig. 9)</td>
</tr>
<tr>
<td>0.001</td>
<td>2.68</td>
<td>1.02</td>
</tr>
<tr>
<td>0.002</td>
<td>4.83</td>
<td>1.49</td>
</tr>
<tr>
<td>0.005</td>
<td>11.0</td>
<td>2.99</td>
</tr>
<tr>
<td>0.01</td>
<td>20.3</td>
<td>5.48</td>
</tr>
<tr>
<td>0.02</td>
<td>35.8</td>
<td>10.2</td>
</tr>
<tr>
<td>0.05</td>
<td>67.3</td>
<td>22.1</td>
</tr>
<tr>
<td>0.1</td>
<td>93.2</td>
<td>36.1</td>
</tr>
</tbody>
</table>
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


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


