A Stochastic Reconfigurable Architecture for Fault-Tolerant Computation with Sequential Logic

Peng Li†, Weikang Qian‡, and David J. Lilja†
†Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, USA
‡University of Michigan-Shanghai Jiao Tong University Joint Institute, Shanghai, China
lipeng@umn.edu, qianwk@sjtu.edu.cn, lilja@umn.edu

***Abstract*—Computation performed on stochastic bit streams is less efficient than that based on a binary radix because of its long latency. However, for certain complex arithmetic operations, computation on stochastic bit streams can consume less energy and tolerate more soft errors. In addition, the latency issue could be solved by using a faster clock frequency or in combination with a parallel processing approach. 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. In this paper, we enhance and extend this reconfigurable architecture using sequential logic. Compared to the previous approach, the proposed reconfigurable architecture takes less hardware area and consumes less energy, while achieving the same performance in terms of processing time and fault-tolerance.**

I. Introduction

Almost all modern computers use binary radix encoding to represent numeric values. It is a positional notation with a radix of 2, and an extremely compact encoding. For example, M symbols can represent 2^M different numeric values. Another encoding scheme, which we call a stochastic encoding scheme [1], represents a numeric value x in the unit interval (i.e., $0 \leq x \leq 1$) by a bit stream X, in which the probability of a one is x. For example, "0.5" could be represented by a bit stream "10001101", in which the probability of a one is "0.5."

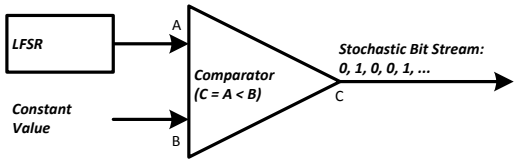

Fig. 1. Binary radix encoding to stochastic bit stream converter [2]. The comparator outputs a one if the random number generated by the LFSR is less than the constant value; it outputs a zero otherwise.

By using a linear feedback shift register (LFSR) and a comparator, we can convert a value from its binary radix encoding to its stochastic encoding. As shown in Fig. 1, assuming that we want to represent x $(0 \leq x \leq 1)$ with an L-bit stochastic stream, we 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. For example, if we want to represent a probability "0.5" with an 8-bit stream, we set the LFSR to generate random numbers in the range $[0, 8)$, and set the constant value to $8 \times 0.5 = 4$. A counter, which counts the number of ones in the stochastic bit stream, can be used to convert the value from its stochastic encoding to its binary radix encoding [2], which is the number of ones divided by the length of the bit stream.

Basic arithmetic operations can be implemented with very simple digital logic circuits with the stochastic encoding scheme [1], [3]. For example, as shown in Fig. 2, multiplication can be implemented using a single *AND* gate. In this figure, the two inputs A and B are independent stochastic bit streams. If we define $a = P(A = 1)$, $b = P(B = 1)$, and $c = P(C = 1)$, based on the logic function of the AND gate, we will have $P(C = 1) = P(A = 1) \cdot P(B = 1)$, i.e., $c = a \cdot b$.

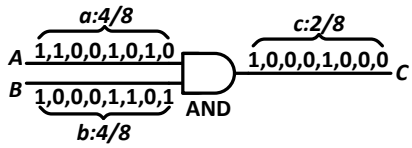

Fig. 2. Multiplication with stochastic encoding. Here the inputs are both $4/8$. The output is $4/8 \times 4/8 = 2/8$, as expected.

It can be seen that the stochastic encoding is a uniform encoding scheme. This is because an M-bit stream can represent only M different numeric values with this encoding scheme. In other words, assuming that a value is represented by M bits using binary radix encoding, we need 2^M bits to represent the same value using stochastic encoding. As a result, for those basic operations such as multiplication and addition, computation on stochastic bit streams could have a long latency and consume more energy.

In previous work, Brown and Card [4], however, showed that for certain complex operations, such as the exponentiation and tanh functions, computation on stochastic bit streams consumes less energy than computation on a binary radix, and the latency issue could be solved by using a parallel processing approach or/and a higher operational frequency thanks to its simple circuit structure [5], [6], [7]. For example, in the applications of artificial neural networks (ANNs) [8] and image processing [9], [10], we can process multiple neurons or pixels at the same time. In addition, researchers also demonstrated that computation on bit streams can tolerate a large number of soft errors [11] compared to computation on binary radix. Qian et al. [2] illustrated the fault-tolerance of computation on stochastic bit streams using the example of the gamma correction function used in image processing. The experimental results showed that, when soft errors are injected at a rate of 15%, the image generated by the computation using a binary radix is full of noisy pixels, while the image generated by the computation on stochastic bit streams is still recognizable.

To take advantage of this computing technique for those complex operations, Qian et al. [2] proposed a reconfigurable architecture using combinational logic. The kernel of their architecture is a generalized multiplexing circuit, which can synthesize a given target function stochastically based on the theory of Bernstein polynomials [12]. We find that sequential logic can also synthesize the given target function stochastically. If we redesign the stochastic reconfigurable architecture using sequential logic, the hardware area and the energy consumption will be significantly reduced, and the performance (in

terms of processing time and fault-tolerance) will still be the same. In this paper, our contributions are: 1) we develop a general approach to synthesize a finite-state machine (FSM) to implement the given target function stochastically; 2) we use this FSM to enhance the stochastic reconfigurable architecture to significantly reduce the hardware area and energy consumption.

The remainder of the paper is organized as follows. Section II briefly reviews the related work. Section III introduces how to synthesize functions stochastically using the FSM. Section IV demonstrates the proposed stochastic reconfigurable architecture based on the FSM. Section V shows the experimental results. Conclusions are drawn in Section VI.

II. Related Work

Logical computation on stochastic bit streams dates back to the 1960s. In an early set of papers, researchers proposed designs to implement basic arithmetic operations such as addition, multiplication, and division on stochastic bit streams [1]. The implementations of more sophisticated functions, such as the tanh and exponentiation functions, have also been proposed [4]. Logical computation on stochastic bit streams finds applications in many different areas, including artificial neural networks (ANNs), communication, control, and image processing [4], [13], [14], [10]. Many earlier works applied logical computation on stochastic bit streams to implement ANNs [15]. In ANNs, we usually require a large number of adders and multipliers. Conventional implementations of adders and multipliers based on binary radix are very costly in area. However, a stochastic implementation of these basic operations is very hardware-efficient, which makes the realization of large scale ANNs possible. Recently, logical computation on stochastic bit streams has also been applied in communication to implement low-density parity-check (LDPC) decoders [13], in control to implement proportional-integral (PI) controller [14], and in image processing for functions such as edge detection, median filtering, and contrast stretching [10].

In 2011, Qian et al. [2] proposed a reconfigurable architecture for performing polynomial computation on stochastic bit streams. This architecture, as shown in Fig. 3, is composed of three parts: the *Randomizer*, the *ReSC Unit*, and the *De-Randomizer*. C_X and C_{Z_i} ($0 \leq i \leq n$, where n is the highest degree of the polynomial this architecture can compute) are the inputs. C_Y is the output. These values are represented using binary radix. The architecture is reconfigurable in the sense that it can be used to compute different functions $C_Y = f(C_X)$ by setting appropriate values for the coefficients C_{Z_i} ($0 \leq i \leq n$) [2].

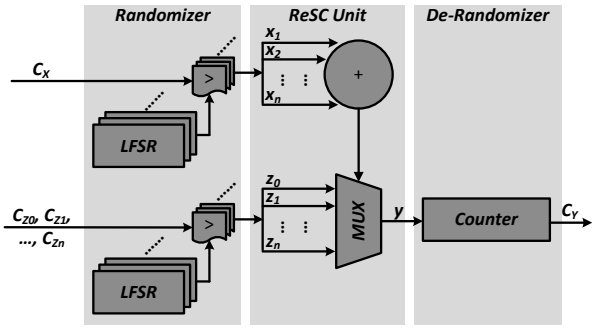

Fig. 3. A reconfigurable stochastic computing architecture based on combinational logic (i.e., the *ReSC Unit* is implemented using an adder and a multiplexer) [2].

Their *Randomizer* uses the circuit shown in Fig. 1 to convert the numerical values C_X and C_{Z_i} to stochastic bit streams X_k ($1 \leq k \leq n$) and Z_i ($0 \leq i \leq n$). Their *De-Randomizer* is implemented using a counter, which converts the resulting bit streams to binary radix encoded values. The *ReSC Unit*, which processes the stochastic bit streams, is the kernel of the architecture. It is a generalized multiplexing circuit which implements Bernstein polynomials [12] 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:

$$f(x) = \frac{2}{8}B_{0,3}(x) + \frac{5}{8}B_{1,3}(x) + \frac{3}{8}B_{2,3}(x) + \frac{6}{8}B_{3,3}(x), \quad (1)$$

where each $B_{i,3}(x)$ ($i = 0, 1, \ldots, 3$) is a Bernstein basis polynomial of the form $B_{i,3}(x) = \binom{3}{i}x^i(1-x)^{3-i}$. A Bernstein polynomial, $B(x) = \sum_{i=0}^{n} b_i B_{i,n}(x)$, with all coefficients b_i in the unit interval, can be implemented stochastically by the *ReSC Unit* shown in Fig. 3. An illustration of how equation (1) is implemented by the *ReSC Unit* is shown in Fig. 4.

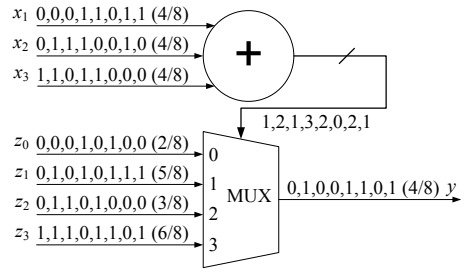

Fig. 4. *ReSC Unit* implementing the Bernstein polynomial $f(x) = \frac{2}{8}B_{0,3}(x) + \frac{5}{8}B_{1,3}(x) + \frac{3}{8}B_{2,3}(x) + \frac{6}{8}B_{3,3}(x)$ at $x = 0.5$. Stochastic bit streams x_1, x_2 and x_3 encode the value $x = 0.5$. Stochastic bit streams z_0, z_1, z_2 and z_3 encode the corresponding Bernstein coefficients [2].

The *ReSC Unit* consists of an adder block and a multiplexer block. The inputs to the adder are an input set $\{x_1, \ldots, x_n\}$. The data inputs to the multiplexer are $z_0, \ldots, z_n$. The outputs of the adder are the selecting inputs to 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 binary number computed by the adder is i and the output of the multiplexer y is set to z_i. The inputs $x_1, \ldots, x_n$ are fed with independent stochastic bit streams $X_1, \ldots, X_n$ representing the probabilities $P(X_i = 1) = x \in [0, 1]$, for $1 \leq i \leq n$. The inputs $z_0, \ldots, z_n$ are fed with independent stochastic bit streams $Z_0, \ldots, Z_n$ representing the probabilities $P(Z_i = 1) = b_i \in [0, 1]$, for $0 \leq i \leq n$, where the b_i's are the Bernstein coefficients. The output of the circuit is a stochastic bit stream Y in which the probability of a bit being one equals the Bernstein polynomial $B(t) = \sum_{i=0}^{n} b_i B_{i,n}(t)$ evaluated at $t = x$.

It can be seen from Fig. 3 that the entire architecture consists of $(2n + 1)$ LFSRs, $(2n + 1)$ comparators, an n-bit adder, an $(n + 1)$-channel multiplexer, and a counter (note that n is the highest degree of the polynomial that this architecture can compute). In this paper, we redesign the *ReSC Unit* using sequential logic to significantly reduce the circuit complexity for the *Randomizer*. The details are discussed in the following sections.

III. Synthesizing Functions Using an FSM

We propose a new circuit shown in Fig. 6 to implement a given target function $T(P_X)$ stochastically. It has the same function as the *ReSC Unit* introduced previously. Thus, this circuit can substitute for

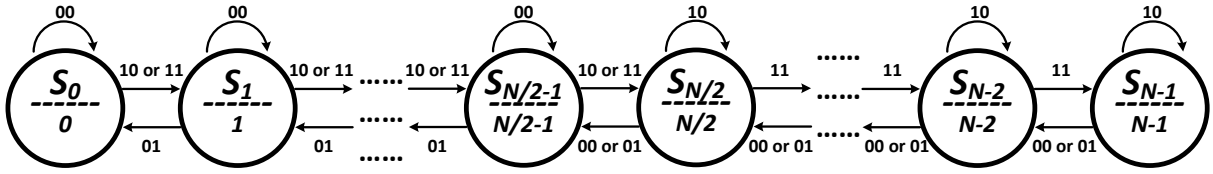

Fig. 5. The state transition diagram of an N-state Moore style FSM. It has two inputs X and K. The numbers on each arrow represent the transition condition, with the first corresponding to the input X and the second corresponding to the input K. This FSM has $\lceil log_2 N \rceil$ outputs, encoding 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.

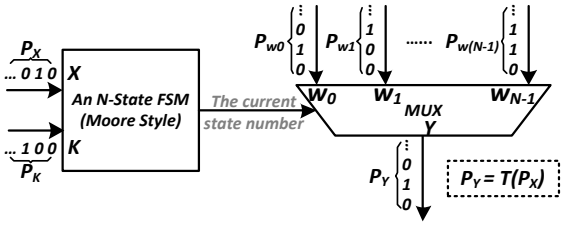

Fig. 6. The new FSM-based circuit for synthesizing target functions stochastically.

the *ReSC Unit* in the reconfigurable architecture shown in Fig. 3, and the hardware cost of the new architecture can be significantly reduced. The new architecture will be introduced in Section IV and its system block diagram can be found in Fig. 9. In this section, we discuss how to use the circuit shown in Fig. 6 to synthesize the given target function $T(P_X)$.

Fig. 5 shows the state transition diagram of the FSM in Fig. 6. It has two inputs X and K. Its output is the current state number. For example, if the current state of the FSM is S_i $(0 \leq i \leq N-1)$, then the output of the FSM is i. Note that although the output of the FSM looks like an up/down counter, its state transition is quite different from a conventional up/down counter. The output of the FSM is connected to the selection bits of the multiplexer "*MUX*", which has N data inputs (w_0, w_1, $\cdots$, w_{N-1}). Note that if the current state of the FSM is S_i, then the "*MUX*" will choose its i-th data input w_i as the output Y.

Assuming that X, K, and w_i are all stochastic bit streams, the output Y is also a stochastic bit stream. We define P_X, P_K, P_{w_i}, and P_Y to be the probabilities of ones in X, K, w_i, and Y, respectively. The synthesis goal is to compute P_{w_i} and P_K to make P_Y approximate the given target function $T(P_X)$. More specifically, we define the approximation error ϵ as follows,

$$\epsilon = \int_0^1 \left(T(P_X) - P_Y\right)^2 \cdot d(P_X). \tag{2}$$

The synthesis goal is to compute P_{w_i} and P_K to minimize ϵ subject to the constraints that $0 \leq P_{w_i} \leq 1$ and $0 \leq P_K \leq 1$. In the following sections, we discuss how to obtain these parameters.

A. Understanding the FSM

Before we introduce the detailed approach for computing P_{w_i} and P_K, it is necessary to understand the state transition diagram of the FSM shown in Fig. 5. Given a current state S_i, the next state of the FSM will be

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

If the inputs X and K 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

$$\begin{cases} P_i \cdot P_X = P_{i+1} \cdot (1 - P_X) \cdot P_K, & 0 \leq i \leq \frac{N}{2} - 2, \\ P_{\frac{N}{2}-1} \cdot P_X = P_{\frac{N}{2}} \cdot (1 - P_X), \\ P_i \cdot P_X \cdot P_K = P_{i+1} \cdot (1 - P_X), & \frac{N}{2} \leq i \leq N - 2, \end{cases} \tag{3}$$

where P_i is the probability that the current state is S_i in the equilibrium state (or the probability that the current output is i). P_X and P_K have been already defined in the beginning of this section. Note that the individual state probability P_i must sum to unity over all S_i, i.e.,

$$\sum_{i=0}^{N-1} P_i = 1. \tag{4}$$

Using (3) and (4), we can write P_i in terms of P_X and P_K as

$$P_i = \begin{cases} \frac{(\frac{P_X}{1-P_X})^i \cdot P_K^{-i}}{\alpha}, & 0 \leq i \leq \frac{N}{2} - 1, \\ \frac{(\frac{P_X}{1-P_X})^i \cdot P_K^{i+1-N}}{\alpha}, & \frac{N}{2} \leq i \leq N - 1, \end{cases} \tag{5}$$

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

Note that based on the circuit shown in Fig. 6, P_Y is a function of P_i and P_{w_i} (we have defined P_Y and P_{w_i} in the beginning of this section):

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

In the next two sections, we will demonstrate how to use (6) to compute P_{w_i} and P_K to synthesize the given target function $T(P_X)$.

B. Compute P_{w_i}

By expanding (2), we can rewrite ϵ as

$$\epsilon = \int_0^1 T(P_X)^2 \cdot d(P_X) - 2\int_0^1 T(P_X) \cdot P_Y \cdot d(P_X) + \int_0^1 P_Y^2 \cdot d(P_X).$$

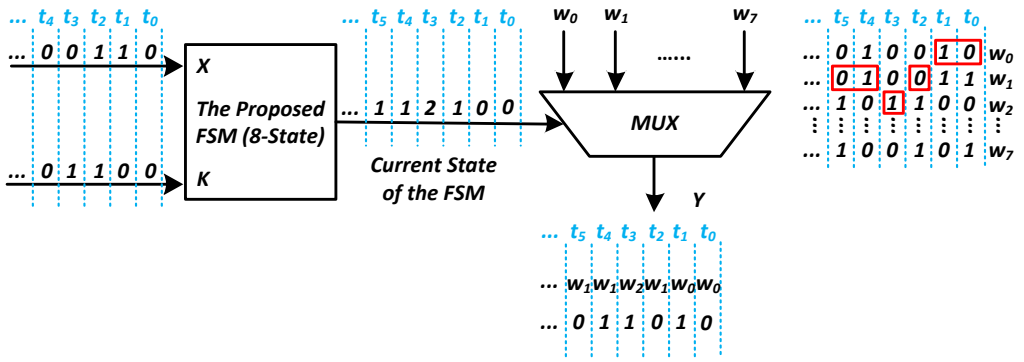

Fig. 7. An illustration showing how the circuit shown in Fig. 6 works based on specific stochastic bit streams of inputs X, K, and w_i ($0 \leq i \leq 7$). The FSM has 8 states. Its state transition has been shown in Fig. 5, and we assume the initial state is S_0. The stochastic bit stream of output Y is generated based on the input bit streams and the corresponding logic circuit.

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

$$\varphi = \int_0^1 P_Y^2 \cdot d(P_X) - 2\int_0^1 T(P_X) \cdot P_Y \cdot d(P_X). \qquad (7)$$

We notice that computing P_{w_i} to minimize φ is a typical constrained quadratic programming problem, if P_K is a constant. When we set P_K to a constant, the integral of P_i on P_X is also a constant. The solution of P_{w_i} can be obtained using standard techniques [16]. The detailed process for a similar problem has been illustrated by Li et al. in [17], [18]. P_K can be solved using a numerical approach, which will be discussed in the next section.

C. Compute P_K

As we stated in the previous section, P_K is first set to a constant. Then we compute P_{w_i} using quadratic programming [16] to minimize ϵ (or the equivalent φ). Note that all the values between 0 and 1 (with a step 0.001) will be used to set P_K in the synthesis process. More specifically, we first set P_K to 0.001, and compute the corresponding P_{w_i} and ϵ. Next, we set P_K to 0.002, and compute the corresponding P_{w_i} and ϵ, and so on. Finally, we set P_K to 1, and compute the corresponding P_{w_i} and ϵ. Among these 1000 results, we select the minimum ϵ, and the corresponding P_K and P_{w_i}.

D. An Example

In this section, we show how the circuit shown in Fig. 6 works based on the example in Equation (1) in Section II (here we use P_X and $T(P_X)$ instead of x and $f(x)$).

Example: Use an 8-state FSM to synthesize:

$$T(P_X) = \frac{1}{4} + \frac{9}{8}P_X - \frac{15}{8}P_X^2 + \frac{5}{4}P_X^3.$$

To implement this function stochastically using the circuit shown in Fig. 6, we first need to compute P_K and P_{w_i}. Based on the proposed synthesis approach, we obtain the parameters shown in Table I. The minimum approximation error ϵ defined in Equation (2) is 6.0×10^{-11}.

TABLE I
P_K AND P_{w_i} FOR SYNTHESIZING THE POLYNOMIAL IN THE ABOVE EXAMPLE USING AN 8-STATE FSM.

$P_K = 0.44$			
$P_{w_0} = 0.25$	$P_{w_1} = 0.73$	$P_{w_2} = 0.16$	$P_{w_3} = 0.70$
$P_{w_4} = 0.30$	$P_{w_5} = 0.84$	$P_{w_6} = 0.26$	$P_{w_7} = 0.75$

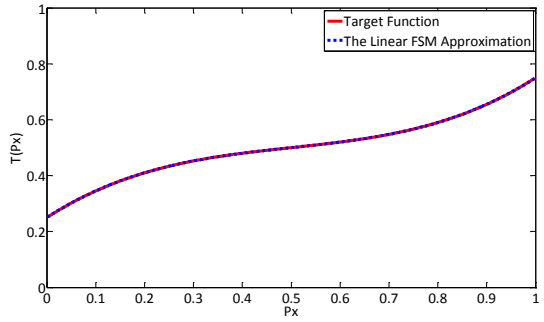

Fig. 8. Simulation result of the example.

Fig. 7 illustrates how the circuit in Fig. 6 works for this example. Assuming that the circuit starts working at clock cycle t_0 and the initial state of the FSM is S_0 (note that the initial state has no influence on the final results, it could be any one of the 8 states), the output of the FSM at t_0 is 0 (because its initial state is S_0) and the output of the multiplexer "*MUX*" at t_0 is $w_0 = 0$ (because its selection input equals 0 at t_0).

Because at t_0, $(X, K) = (0, 0)$ and the initial state is S_0, in the next clock cycle t_1, the output of the FSM is still S_0 based on the state transition diagram shown in Fig. 5. The output of the "*MUX*" at t_1 is still w_0, and $w_0 = 1$ at t_1.

In the next clock cycle t_2, the state becomes S_1 because $(X, K) = (1, 0)$ at the previous clock cycle t_1, and the output of the "*MUX*" at t_2 becomes w_1, which equals 0 at t_2, and so on for the other clock cycles.

Assume that we use 1024 bits to represent a value stochastically. After 1024 clock cycles, if the probability of ones in X equals P_X, the probability of ones in K equals 0.44, and the probability of ones in w_i equals P_{w_i}, shown in Table I, then the probability of ones in

Y will be

$$P_Y \approx \frac{1}{4} + \frac{9}{8}P_X - \frac{15}{8}P_X^2 + \frac{5}{4}P_X^3.$$

Fig. 8 shows the simulation of this circuit with P_X ranging from 0 to 1.

Another example is the linear gain function proposed by Brown and Card [4]. If we set the corresponding target function, we will get exactly the same results (i.e., $P_{w_i} = 0$ if $0 \le i \le \frac{N}{2} - 1$; else $P_{w_i} = 1$). In addition, based on the expression of P_i in (5), we find that an N-state FSM could be used to synthesize the polynomials with a degree up to $N - 1$. Thus, the 8-state FSM can be used to synthesize polynomials with a degree up to 7, which is high enough for most applications. As a result, we use the 8-state FSM to construct the proposed stochastic reconfigurable architecture.

IV. The Proposed Stochastic Reconfigurable Architecture Using the FSM

The proposed reconfigurable architecture using the FSM is shown in Fig. 9. Similar to the original reconfigurable architecture shown in Fig. 3, it is also composed of three parts: the *Randomizer*, the *FSM Unit*, and the *De-Randomizer*. The inputs are C_X, C_K, and C_{w_i} ($0 \le i \le 7$). C_Y is the output. These values are represented using binary radix. The architecture is reconfigurable in the sense that it can be used to compute different functions $C_Y = f(C_X)$ by setting appropriate values for the constants C_K and C_{w_i} ($0 \le i \le 7$).

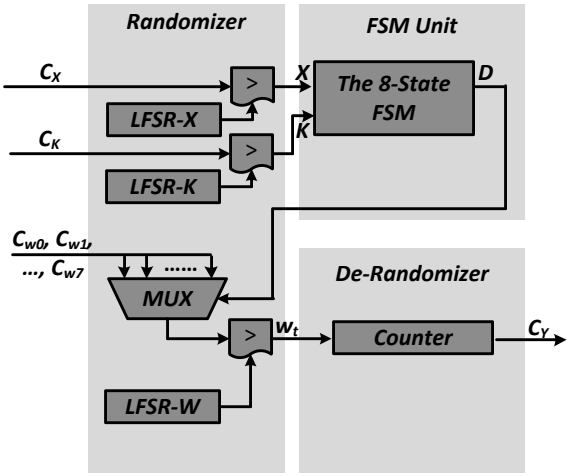

Fig. 9. An FSM-based reconfigurable stochastic architecture.

The *Randomizer* has the same function as shown in Fig. 3, which converts the numerical values C_X, C_K and C_{w_i} to the corresponding stochastic bit streams. However, it takes much less hardware than the combinational implementation. As we described in Section II, the original *Randomizer* uses $(2n+1)$ LFSRs and $(2n+1)$ comparators in total (n for C_X, $(n+1)$ for C_{Z_i}, as shown in Fig. 3), where n is the degree of the target polynomial. In the proposed architecture, the *Randomizer* uses only 3 LFSRs, 3 comparators, and an 8-channel multiplexer. Because the LFSR is the most area consuming part in the entire architecture, the *Randomizer* in the new architecture saves $(2n-2)$ LFSRs compared to the original *ReSC Unit*.

The *FSM Unit* has a similar function as the *ReSC Unit* in the original architecture. It contains only an 8-state FSM which is implemented based on the state transition diagram shown in Fig. 5. Note that if the current state of this FSM is S_i ($0 \le i \le 7$), then the *MUX* in the *Randomizer* will connect its i-th data input (i.e., C_{w_i}) to the output of the *MUX*, and will generate the corresponding bit using the *LFSR-W* and the comparator. This implementation essentially has the same behavior as the circuit shown in Fig. 6, which needs n LFSRs and n comparators to generate n different stochastic bit streams with probabilities of ones being $P_{w_0}, P_{w_1}, \cdots, P_{w_{n-1}}$, respectively. We notice that at each clock cycle one of the n random input bits to the *MUX* 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 (shown in Fig. 1) according to the current state number. Thus, we use the output of the FSM to choose from the constant values $C_{w_0}, C_{w_1}, \cdots, C_{w_{n-1}}$. Note that in order to multiplex constant values of m bits, the *MUX* in Fig. 9 stands for an 8-channel m-bit multiplexer.

The *De-Randomizer* is implemented using a binary counter, which converts the resulting bit streams to output values. This is the same as the original one shown in Fig. 3.

TABLE II
C_K and C_{w_i} for computing the polynomial example.

$C_K = 448$			
$C_{w_0} = 256$	$C_{w_1} = 748$	$C_{w_2} = 164$	$C_{w_3} = 717$
$C_{w_4} = 307$	$C_{w_5} = 860$	$C_{w_6} = 266$	$C_{w_7} = 768$

To illustrate how this architecture works, we use the same example introduced in Section III-D. A numerical value is represented by a stochastic 1024-bit stream. We set the constants C_K and C_{w_i} based on P_K and P_{w_i} given in Table I. The values of C_K and C_{w_i} are shown in Table II, where we have $C_K = P_K \times 1024$ and $C_{w_i} = P_{w_i} \times 1024$. This means that, using the circuit shown in Fig. 9, if we set C_K and C_{w_i} ($0 \le i \le 7$) to the corresponding values shown in Table II, C_Y will approximate the following function,

$$\frac{C_Y}{1024} = \frac{1}{4} + \frac{9}{8} \times \left(\frac{C_X}{1024}\right) - \frac{15}{8} \times \left(\frac{C_X}{1024}\right)^2 + \frac{5}{4} \times \left(\frac{C_X}{1024}\right)^3.$$

V. Experimental Results

In this section, we compare the proposed stochastic reconfigurable architecture to the one proposed by Qian et al. [2] in terms of the hardware area and the fault-tolerance.

A. Hardware Area and Energy Consumption Comparison

It can be seen from Fig. 3 and Fig. 9 that the two architectures have several basic units in common, such as the LFSR, the comparator, and the counter. Since we use 1024 (i.e., 2^{10}) bits to represent a numerical value stochastically, the bit width of these components is 10. We use the Synopsys Design Compiler to evaluate the hardware area of these basic components in terms of equivalent fanin-two NAND gates. We find that a 10-bit LFSR takes 60 gates, a 10-bit comparator takes 30 gates, and a 10-bit counter takes 60 gates. In addition, an n-bit adder takes at least $2n$ gates, and an n-bit multiplexer takes $3(n-1)$ gates. The 8-state FSM takes 28 gates.

Assuming that the target polynomial has a degree of n, the hardware area of the stochastic reconfigurable architecture proposed by Qian et al. [2] depends on n. As we introduced in Fig. 3, its *Randomizer* consists of $2n+1$ LFSRs and $2n+1$ comparators. Its *R*eSC Unit consists of an n-bit adder and an $(n+1)$-channel multiplexer. Its *De-Randomizer* is implemented using a counter. As a result, its total hardware area is $(185n + 150)$ gates.

In contrast, the hardware area of the proposed stochastic reconfigurable architecture is independent of the degree of the target

polynomial as long as the degree is less than or equal to 7. Note that if the degree is greater than 7, we need an FSM with more states, such as a 16-state FSM, to implement the target polynomial. As shown in Fig. 9, the *Randomizer* in the proposed architecture consists of 3 LFSRs, 3 comparators, and an 8-channel 10-bit multiplexer. Its *FSM Unit* consists of the 8-state FSM. Its *De-Randomizer* is a counter. As a result, its hardware area is only 568 gates. We summarize the hardware area of these two architectures versus the degree of the target polynomials in Table III. It can be seen that the proposed architecture takes much less hardware area than the one proposed by Qian et al. [2]. For example, when $n = 7$, it takes only 39% of the area of the one proposed by Qian et al. [2].

Since both architectures compute on stochastic bit streams, they have the same processing time for the same operations. We normally evaluate the energy consumption by the product of the processing time and the hardware area. However, because the proposed architecture takes much less hardware area while keeping the computation time constant, it also consumes much less energy. Compared to conventional hardware implementations based on a binary radix, the proposed stochastic reconfigurable architecture has smaller area-delay product as was shown previously when comparing the *Re*SC architecture to implementations based on binary radix [2].

TABLE III
AREA COMPARISON FOR POLYNOMIALS OF DEGREES FROM 3 TO 7.

Degree n	3	4	5	6	7
Qian et. al [2]	705	890	1075	1260	1445
Proposed	568				

B. Fault-Tolerance Comparison

We compare the performance of stochastic computation on polynomial functions when the input data are corrupted with noise. Suppose that the bit stream of a stochastic representation contains $2^{10} = 1024$ bits. We choose the error ratio λ of the input data to be 0, 0.001, 0.005, 0.01, 0.05, and 0.1, as measured by the fraction of random bit flips that occur.

We randomly choose 10 polynomials of degree between 3 and 7 with coefficients in the unit interval. We evaluated each target function on 10 points: 0.1, 0.2, 0.3, $\cdots$, 0.9, 1. For each error ratio λ, each target function, and each evaluation point, we simulated both the combinational logic-based stochastic reconfigurable architecture and the FSM-based stochastic reconfigurable architecture 1000 times. We averaged the relative errors over all simulations. Finally, for each error ratio λ, we averaged the relative errors over all evaluation points and all target polynomials. Table IV shows the average relative error of the two stochastic reconfigurable architectures versus different error ratios λ. It can be seen that the two stochastic reconfigurable architectures have almost the same performance in terms of fault-tolerance (the difference is within 0.3%). This is because both of them perform computation on stochastic bit streams, which can tolerate more errors than those based on a binary radix [2] .

TABLE IV
RELATIVE ERROR FOR THE COMBINATIONAL LOGIC-BASED STOCHASTIC RECONFIGURABLE ARCHITECTURE AND THE FSM-BASED STOCHASTIC RECONFIGURABLE ARCHITECTURE VERSUS THE ERROR RATIO λ IN THE INPUT DATA. THE DIFFERENCE IS WITHIN 0.3%.

Error ratio λ	0	0.001	0.005	0.01	0.05	0.1
*Re*SC (%)	2.63	2.62	2.73	3.01	7.54	13.8
*F*SM (%)	2.65	2.71	2.92	2.83	7.68	14.1

VI. CONCLUSION AND FUTURE WORK

This paper describes a reconfigurable architecture to perform computation on stochastic bit streams using sequential logic. Compared to the previous stochastic reconfigurable architecture based on combinational logic, it has the same performance in terms of fault-tolerance and processing time, but takes less hardware. Future research will work on a general purpose stochastic processor.

ACKNOWLEDGMENT

This work was supported in part by National Science Foundation grant no. CCF-1241987. 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. This work was also supported in part by the Minnesota Supercomputing Institute and a donation from NVIDIA.

REFERENCES

[1] B. Gaines, "Stochastic computing systems," *Advances in Information Systems Science*, vol. 2, no. 2, pp. 37–172, 1969.

[2] W. Qian, X. Li, M. Riedel, K. Bazargan, and D. Lilja, "An architecture for fault-tolerant computation with stochastic logic," *Computers, IEEE Transactions on*, vol. 60, no. 1, pp. 93–105, 2011.

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

[4] B. Brown and H. Card, "Stochastic neural computation. I. Computational elements," *Computers, IEEE Transactions on*, vol. 50, no. 9, pp. 891–905, 2001.

[5] M. Alioto, G. Palumbo, and M. Pennisi, "Understanding the effect of process variations on the delay of static and domino logic," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 18, no. 5, pp. 697–710, 2010.

[6] Y. Li, N. Zeng, W. Hung, and X. Song, "Enhanced symbolic simulation of a round-robin arbiter," in *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pp. 102–107, IEEE, 2011.

[7] C. Liu, J. Su, and Y. Shi, "Temperature-aware clock tree synthesis considering spatiotemporal hot spot correlations," in *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pp. 107–113, IEEE, 2008.

[8] A. Morgenstern and K. Schneider, "Synthesizing deterministic controllers in supervisory control," *Informatics in control, automation and robotics II*, pp. 95–102, 2007.

[9] P. Li and D. Lilja, "A low power fault-tolerance architecture for the kernel density estimation based image segmentation algorithm," in *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*, pp. 161–168, IEEE, 2011.

[10] P. Li and D. Lilja, "Using stochastic computing to implement digital image processing algorithms," in *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pp. 154–161, IEEE, 2011.

[11] J. Rivers, P. Bose, P. Kudva, J. Wellman, P. Sanda, E. Cannon, and L. Alves, "Phaser: Phased methodology for modeling the system-level effects of soft errors," *IBM Journal of Research and Development*, vol. 52, no. 3, pp. 293–306, 2008.

[12] G. Lorentz, *Bernstein Polynomials*. University of Toronto Press, 1953.

[13] S. Sharifi Tehrani, S. Mannor, and W. Gross, "Fully parallel stochastic LDPC decoders," *Signal Processing, IEEE Transactions on*, vol. 56, no. 11, pp. 5692–5703, 2008.

[14] D. Zhang and H. Li, "A stochastic-based FPGA controller for an induction motor drive with integrated neural network algorithms," *Industrial Electronics, IEEE Transactions on*, vol. 55, no. 2, pp. 551–561, 2008.

[15] J. Tomberg and K. Kaski, "Pulse-density modulation technique in VLSI implementations of neural network algorithms," *Solid-State Circuits, IEEE Journal of*, vol. 25, no. 5, pp. 1277–1286, 1990.

[16] G. Golub and C. Van Loan, *Matrix computations*, vol. 3. Johns Hopkins Univ Pr, 1996.

[17] P. Li, D. J. Lilja, W. Qian, and K. Bazargan, "Using a two-dimensional finite-state machine for stochastic computation," in *International Workshop on Logic and Synthesis, IWLS'12*, 2012.

[18] P. Li, D. J. Lilja, W. Qian, K. Bazargan, and M. Riedel, "The synthesis of complex arithmetic computation on stochastic bit streams using sequential logic," in *Computer-Aided Design, 2012. ICCAD 2012. IEEE/ACM International Conference on*, IEEE, 2012.