The Synthesis of Robust Polynomial Arithmetic with Stochastic Logic

Weikang Qian and Marc D. Riedel
Department of Electrical and Computer Engineering
University of Minnesota, Twin Cities
{qianx030, mriedel}@umn.edu

ABSTRACT

As integrated circuit technology plumbs ever greater depths in the scaling of feature sizes, maintaining the paradigm of deterministic Boolean computation is increasingly challenging. Indeed, mounting concerns over noise and uncertainty in signal values motivate a new approach: the design of stochastic logic, that is to say, digital circuitry that processes signals *probabilistically*, and so can cope with errors and uncertainty. In this paper, we present a general methodology for synthesizing stochastic logic for the computation of polynomial arithmetic functions, a category that is important for applications such as digital signal processing. The method is based on converting polynomials into a particular mathematical form – Bernstein polynomials – and then implementing the computation with stochastic logic. The resulting logic processes serial or parallel streams that are random at the bit level. In the aggregate, the computation becomes *accurate*, since the results depend only on the precision of the statistics. Experiments show that our method produces circuits that are *highly tolerant of errors* in the input stream, while the area-delay product of the circuit is comparable to that of deterministic implementations.

Categories and Subject Descriptors

B.6.1 [**Logic Design**]: Design Styles—*Combinational Logic, Stochastic Logic*; B.8.1 [**Performance and Reliability**]: Reliability, Testing and Fault-Tolerance

General Terms

Design, Performance, Reliability

Keywords

Stochastic Logic, Probabilistic Logic, Polynomial Arithmetic

1. INTRODUCTION

The successful paradigm for integrated circuit design has been to maintain a sharp boundary in abstraction between the physical and logical layers. From the logic level up, the computation consists of a deterministic sequence of zeros and ones. The precise Boolean functionality of a circuit is prescribed; it is up to the physical layer to produce voltage values that can be interpreted as the exact logical values that are called for. This abstraction is firmly entrenched yet costly: variability, uncertainty, noise – all must be compensated for through ever more complex design and manufacturing. As technology continues to scale, with mounting concerns over noise and uncertainty in signal values, the cost of the abstraction is becoming untenable.

We are developing a framework for digital IC design based on the concept of *stochastic logic*. This paradigm has been known in the literature for many years [7]. Instead of computing with deterministic signals, operations at the logic level are performed on random serial or parallel bit streams. The streams are digital, consisting of zeros and ones; they are processed by ordinary logic gates, such as AND and OR. However, they convey values through the *statistical distribution* of the logical values. Real values in the interval [0, 1] correspond to the *probability of occurrence* of logical one versus logical zero in an observation interval. In this way, computations in the deterministic Boolean domain are transformed into probabilistic computations in the real domain.

Stochastic logic has the advantage that basic arithmetic operations can be performed with simple logic circuits [3]. It suffers from small estimation errors due to the inherent variance in the stochastic bit streams; however, this does not hinder its applications in areas like artificial neural networks and image processing where some inaccuracy can be tolerated [1, 2, 5]. Previous work has shown how basic arithmetic operations like multiplication, addition, and division can be implemented with stochastic logic [3, 8, 10]. In this work, we study the topic more broadly.

First, we present a result in *analysis*. In Section 2, we show that the stochastic behavior obtained from any combinational circuit with random Boolean inputs corresponds to the computation of a multivariate polynomial of a specific form (one with integer coefficients and with the degree of each variable at most 1). Next, we present a methodology for *synthesis*. In Section 3, we describe how to implement arbitrary univariate polynomials with stochastic logic. These are obtained from multivariate polynomials by associating some of the inputs with independent copies of a single random variable and fixing others to be real constants. The synthesis method is based on converting the polynomials into a particular mathematical form, namely Bernstein polynomials.

Our approach is applicable for polynomial-arithmetic circuits that must cope with noise and uncertainty in their inputs. If noise-related faults produce random bit flips in the input streams, these result in fluctuations in the statistics; accuracy can be regained through increased redundancy. In Section 4, we present experimental results on both the measurements of the cost and the error-tolerance of the resulting polynomial-arithmetic circuits. These results show that our method produces circuits that are highly tolerant of errors, while the area-delay product of the circuit is comparable to

DAC 2008, June 8–13, 2008, Anaheim, California, USA.

that of deterministic implementations.

2. STOCHASTIC LOGIC

Our approach is based on a novel view of how to design circuits characterized by noise, variation and uncertainty: instead of transforming Boolean values into Boolean values, such circuits transform *probability* values into *probability* values. The inputs to the circuit are random Boolean variables; the outputs are also random Boolean variables. The computation transforms a *probability distribution* on its inputs to one on its outputs.

2.1 Mathematical Model

In this work, we only consider combinational circuitry. We call combinational logic with random Boolean inputs *stochastic logic*.

Assume that a combinational circuit implements the Boolean function $y = f(x_1, x_2, \ldots, x_n)$. Let $X_1, X_2, \ldots, X_n$ be n *independent* random variables, each with a Bernoulli distribution, and assume that the probability of a logical one for X_i is p_{X_i}. We write $P(X_i = 1) = p_{X_i}$ and $P(X_i = 0) = 1 - p_{X_i}$.

With these random variables as inputs, the output is also a random variable $Y = f(X_1, X_2, \ldots, X_n)$ with a Bernoulli distribution. Let the probability of a logical one for Y be p_Y. We write $P(Y = 1) = p_Y$ and $P(Y = 0) = 1 - p_Y$.

Evidently, p_Y is uniquely determined by the given n-tuple $(p_{X_1}, \ldots, p_{X_n})$. In fact, p_Y is given by an *integer-coefficient multivariate polynomial* on the arguments $p_{X_1}, \ldots, p_{X_n}$. To see this, first note that p_Y is the sum of the probability of occurrence of all combinations of input values for which the Boolean function evaluates to 1. That is,

$$\begin{aligned} p_Y &= P(Y = 1) \\ &= \sum_{\substack{x_1,\ldots,x_n: \\ f(x_1,\ldots,x_n)=1}} P(X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n). \end{aligned}$$

Since $X_1, X_2, \ldots, X_n$ are independent, we further have

$$p_Y = \sum_{\substack{x_1,\ldots,x_n: \\ f(x_1,\ldots,x_n)=1}} \left(\prod_{k=1}^{n} P(X_k = x_k) \right). \tag{1}$$

Since $P(X_i = x_i)$ is either p_{X_i} or $1 - p_{X_i}$, depending on the value of x_i in the given combination, it is easily seen that p_Y is a multivariate polynomial with arguments $p_{X_1}, p_{X_2}, \ldots, p_{X_n}$. Moreover, if we expand Equation (1) into a power form, each product term has an integer coefficient and the degree of each variable in that term is less than or equal to 1.

Thus, we have the following theorem describing the general form of any function that can be computed by stochastic logic.

Theorem 1
Stochastic logic computes a multivariate polynomial of the form

$$\begin{aligned} p_Y &= F(p_{X_1}, p_{X_2}, \ldots, p_{X_n}) \\ &= \sum_{i_1=0}^{1} \cdots \sum_{i_n=0}^{1} \left(\alpha_{i_1 \ldots i_n} \prod_{k=1}^{n} p_{X_k}^{i_k} \right), \end{aligned} \tag{2}$$

where the $\alpha_{i_1 \ldots i_n}$'s are integer coefficients. □

As an example, consider the computation performed by a multiplexer, shown in Figure 2(b). The Boolean function of the multiplexer is

$$y = f(x_1, x_2, s) = (x_1 \wedge s) \vee (x_2 \wedge \neg s),$$

where $\wedge$ means logical AND, $\vee$ means logical OR, and $\neg$ means logical negation. By the definition of p_Y, we have

$$\begin{aligned} p_Y &= P(X_1 = 1, X_2 = 0, S = 1) + P(X_1 = 1, X_2 = 1, S = 1) \\ &\quad + P(X_1 = 0, X_2 = 1, S = 0) + P(X_1 = 1, X_2 = 1, S = 0) \\ &= p_{X_1}(1 - p_{X_2})p_S + p_{X_1}p_{X_2}p_S \\ &\quad + (1 - p_{X_1})p_{X_2}(1 - p_S) + p_{X_1}p_{X_2}(1 - p_S) \\ &= p_{X_2} + p_{X_1}p_S - p_{X_2}p_S, \end{aligned} \tag{3}$$

which confirms that p_Y is an integer-coefficient multivariate polynomial on the arguments p_{X_1}, p_{X_2}, and p_S. The degree of each variable in each product term is less than or equal to 1.

2.2 Implementation

Given a combinational circuit implementing a Boolean function $y = f(x_1, x_2, \ldots, x_n)$, stochastic logic is implemented as follows. We generate n independent stochastic bit streams $X_1, X_2, \ldots, X_n$, each consisting of N bits. Each bit in the stream X_i equals 1 with independent probability p_{X_i}. The stream is fed into the corresponding input x_i. Thus, in a statistical sense, each bit stream represents a random Boolean variable. In this way, when we measure the rate of the occurrence of 1 in the output bit stream, it gives us an estimate of p_Y. If the bit stream is sufficiently long, this estimate becomes accurate. We assume that the input and the output of the circuit are directly usable in this form. For instance, in sensor applications, analog voltage discriminating circuits might be used to transform real-valued input and output values into and out of probabilistic bit streams.

These bit streams may be *serial* or *parallel*. In serial streams, the random bits arrive sequentially in time. For parallel streams, we make N identical copies of the combinational circuit and feed independent random bits to each copy simultaneously. The choice between serial and parallel stochastic logic translates into a trade-off between time and area.

Figure 1 illustrates a serial implementation with two inputs and one output. The inputs and output are stochastic bit streams that are 8 bits in length. For the output, there are four 1's out of a total of 8 bits. Thus, the estimate is $p_Y = 4/8 = 0.5$.

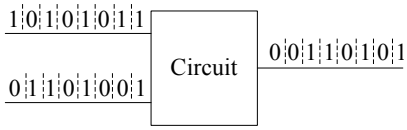

Figure 1: A serial implementation of stochastic logic with inputs and outputs as serial bit streams.

While the method entails redundancy in the encoding of signal values, complex operations can be performed using simple logic.

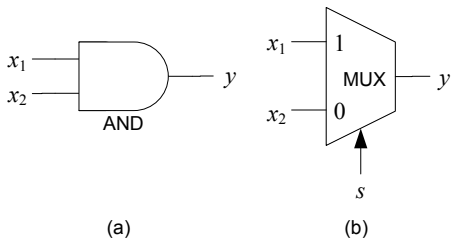

Figure 2: Implementation of multiplication and scaled addition.

- **Multiplication**: consider a two-input AND gate, shown in Figure 2(a). Let its inputs be x_1 and x_2. Then, we have

$$p_Y = P(X_1 = 1, X_2 = 1) = p_{X_1} \cdot p_{X_2}.$$

Thus, the AND gate computes the product.

- **Scaled Addition**: consider a two-input multiplexer, shown in Figure 2(b). Its inputs are x_1 and x_2 and its selecting input is s. From Equation (3), we have

$$p_Y = p_S p_{X_1} + (1 - p_S) p_{X_2},$$

which corresponds to scaled addition.

3. SYNTHESIS OF STOCHASTIC LOGIC FOR POLYNOMIAL ARITHMETIC

Stochastic logic generally implements a special type of multivariate polynomial on input arguments, as was shown by Theorem 1. If we associate some of the p_{X_i}'s of the polynomial $F(p_{X_1}, p_{X_2}, \ldots, p_{X_n})$ in Equation (2) with real constants in the unit interval and the others with a common variable t, then the function F becomes a real-coefficient univariate polynomial $g(t)$. For example, if we set $p_{X_1} = 0.2$, $p_{X_2} = 0.8$, and $p_S = t$ in Equation (3), then we get $g(t) = 0.8 - 0.6t$. With different choices of the original Boolean function f and different settings of the probabilities p_{X_i}'s, we get different polynomials $g(t)$.

In many applications, we need to perform a specific polynomial computation. If given an arbitrary polynomial, how can we synthesize stochastic logic to implement this computation? In this work, we propose a synthesis method that entails converting polynomials from a general power-form representation into a specific form, called *Bernstein polynomials* [9].

3.1 Bernstein Polynomials

Definition 1
The family of $n+1$ polynomials in the form

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}, \qquad i = 0, \ldots, n$$

are called Bernstein basis polynomials *of degree n.*

Definition 2
A linear combination of Bernstein basis polynomials of degree n

$$B^n(t) = \sum_{i=0}^{n} b_i^n B_i^n(t) \tag{4}$$

is a Bernstein polynomial *of degree n. The b_i^n's are called* Bernstein coefficients.

Polynomials of interest are usually represented in *power form*. We can convert a power-form polynomial of degree n, $g(t) = \sum_{i=0}^{n} a_i^n t^i$, into a Bernstein polynomial of degree n as $g(t) = \sum_{i=0}^{n} b_i^n B_i^n(t)$. The conversion from power-form coefficients a_i^n to Bernstein coefficients b_i^n is described in [6]:

$$b_i^n = \sum_{j=0}^{i} \frac{\binom{i}{j}}{\binom{n}{j}} a_j^n, \quad 0 \le i \le n. \tag{5}$$

Generally, a power-form polynomial of degree n can be converted into an equivalent Bernstein polynomial of degree greater than or equal to n. The coefficients of a Bernstein polynomial of degree $m+1$ ($m \ge n$) can be derived from the Bernstein coefficients of an equivalent Bernstein polynomial of degree m. Again, this is described in [6]:

$$b_i^{m+1} = \begin{cases} b_0^m & i = 0 \\ (1 - \frac{i}{m+1}) b_i^m + \frac{i}{m+1} b_{i-1}^m & 1 \le i \le m \\ b_m^m & i = m+1 \end{cases} \tag{6}$$

3.2 Stochastic Logic Computing Bernstein Polynomials with Coefficients in the Unit Interval

If all the coefficients of a Bernstein polynomial are in the unit interval, i.e., $0 \le b_i^n \le 1$, for all $0 \le i \le n$, then we can build stochastic logic to implement it. Figure 3 shows the block diagram of the circuit.

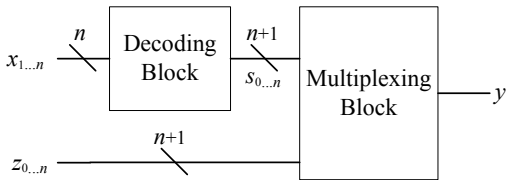

Figure 3: Stochastic logic computing a Bernstein polynomial with coefficients in the unit interval.

The decoding block in Figure 3 has n inputs $x_1, x_2, \ldots, x_n$ and $n+1$ outputs $s_0, s_1, \ldots, s_n$. If i ($0 \le i \le n$) out of the n inputs of the decoding block are logical 1, then s_i is set to 1 and the other outputs are set to 0. Figure 4 gives the implementation of the decoding block with 8 inputs. The eight inputs are grouped into 4 pairs and each pair is fed into a 1-bit adder, which gives a 2-bit sum as the output. The 4 sets of outputs of the 1-bit adder are further grouped into 2 pairs and each pair is fed into a 2-bit adder, which gives a 3-bit sum as the output. The pair of outputs of the 2-bit adder are fed into a 3-bit adder, which gives a 4-bit sum as the output. Finally, the 4-bit sum is fed into a 4-to-9 decoder. Given an input binary number equal to k ($0 \le k \le 8$), the decoder has its k-th output set to 1 and its other outputs set to 0. The output of the decoder gives the output signal s_i's. A decoding block with n inputs can be implemented in a similar way.

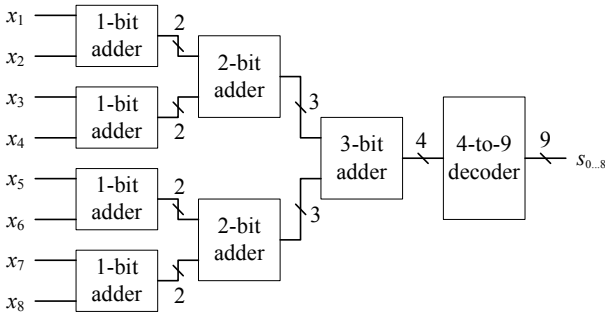

Figure 4: The implementation of the decoding block.

Assume that a k-bit adder has area and depth both proportional to k. Since there are no more than $\lceil n/2 \rceil$ 1-bit adders, no more than $\lceil n/4 \rceil$ 2-bit adders, and so on, the area complexity of the adder tree in the decoding block with n inputs is

$$A_{ar}(n) \le \sum_{i=1}^{\lceil \log_2 n \rceil} O(1) i \left\lceil \frac{n}{2^i} \right\rceil = O(n),$$

and the depth complexity of the adder tree is

$$D_{ar}(n) = \sum_{i=1}^{\lceil \log_2 n \rceil} O(1)i = O(\lg^2 n).$$

The decoder in the decoding block with n inputs is a $\lceil \log_2(n+1) \rceil$-to-$n$ decoder. Assume that we use fanin-2 gates. Then, the area complexity of the decoder is $A_{dr}(n) = O(n)$; the depth complexity is $D_{dr}(n) = O(\lg n)$.

The outputs of the decoding block are further fed into a multiplexing block, as shown in Figure 3, and act as the selecting signals. The data signals of the multiplexing block consist of $n+1$ inputs $z_0, \ldots, z_n$. The Boolean logic of the multiplexing block is

$$y = \bigvee_{i=0}^{n} (z_i \wedge s_i), \tag{7}$$

which means that the output of the multiplexing block y is set to be the input z_i if $s_i = 1$. The area complexity of the multiplexing block is $A_{mb}(n) = O(n)$; the depth complexity is $D_{mb}(n) = O(\lg n)$. (Again, we are assuming that we are using fanin-2 gates.) Thus, the area complexity of the whole circuit is $A(n) = A_{ar}(n) + A_{dr}(n) + A_{mb}(n) = O(n)$; the depth complexity is $D(n) = D_{ar}(n) + D_{dr}(n) + D_{mb}(n) = O(\lg^2 n)$.

Let $X_1, \ldots, X_n$ be n independent Boolean random variables that are 1 with probability $p_{X_i} = t$ $(1 \le i \le n)$. Feed these signals to the corresponding inputs of the decoding block. Let S_i denote the corresponding random output of the decoding block. Since s_i is set to 1 if and only if i out of n inputs of the decoding block are 1, the probability that S_i is 1 is

$$P(S_i = 1) = \binom{n}{i} t^i (1-t)^{n-i} = B_i^n(t), \ 0 \le i \le n. \tag{8}$$

Let $Z_0, \ldots, Z_n$ be $n+1$ independent Boolean random variables that are 1 with probability $p_{Z_i} = b_i^n$ $(0 \le i \le n)$. Feed these signals to the corresponding data inputs of the multiplexing block. Notice that we can set the probability value to be b_i^n because we assume that $0 \le b_i^n \le 1$. Let Y denote the output random Boolean variable. The probability that Y is 1 is

$$p_Y = P(Y=1) = \sum_{i=0}^{n} \left(P(Y=1|S_i=1) P(S_i=1) \right). \tag{9}$$

Since when $S_i = 1$, Y equals Z_i, we have

$$P(Y=1|S_i=1) = P(Z_i=1) = b_i^n. \tag{10}$$

Thus, from Equations (4), (8), (9), and (10), we have

$$p_Y = \sum_{i=0}^{n} b_i^n B_i^n(t) = B^n(t), \tag{11}$$

which means that the circuit in Figure 3 with specific random variables as inputs implements the given Bernstein polynomial with coefficients in the unit interval. Thus, we have the following theorem.

Theorem 2
If all the coefficients of a Bernstein polynomial are in the unit interval, i.e., $0 \le b_i^n \le 1$, for $0 \le i \le n$, then we can design stochastic logic to compute the Bernstein polynomial. □

3.3 Synthesis of Stochastic Logic to Compute Power-Form Polynomials

A polynomial is generally represented in a power form. If it can be converted into a Bernstein polynomial with coefficients in the unit interval, then the preceding section tells us how to implement it with stochastic logic. So what kind of polynomials can be represented as Bernstein polynomials with coefficients in the unit interval? The following theorem gives the answer.

Theorem 3
If $g(t)$ is a polynomial such that

1. *$g(t)$ is identically equal to 0 or to 1, or*
2. *$g(t)$ is strictly greater than 0 and less than 1, for $0 < t < 1$, and both $g(0)$ and $g(1)$ are greater than or equal to 0 and less than or equal to 1, i.e., $0 < g(t) < 1, \forall t \in (0,1)$ and $0 \le g(0), g(1) \le 1$,*

then $g(t)$ can be converted into a Bernstein polynomial of degree m with coefficients $0 \le b_i^m \le 1$ $(i = 0, 1, \ldots, m)$. □

We omit the proof.

We should note here that the degree of the equivalent Bernstein polynomial with coefficients in the unit interval may be greater than the degree of the original polynomial. For example, consider the polynomial $g(t) = 3t - 8t^2 + 6t^3$ of degree 3, satisfying the condition $g(t) \in (0,1), \forall t \in (0,1)$ and $g(0) = 0, g(1) = 1$. Since

$$g(t) = B_1^3(t) - \frac{2}{3}B_2^3(t) + B_3^3(t) = \frac{3}{4}B_1^4(t) + \frac{1}{6}B_2^4(t) - \frac{1}{4}B_3^4(t) + B_4^4(t) = \frac{3}{5}B_1^5(t) + \frac{2}{5}B_2^5(t) + B_5^5(t),$$

the degree of the equivalent Bernstein polynomial with coefficients in the unit interval is 5.

Based on Theorems 2 and 3, we have the following corollary, which gives a *sufficient* condition:

Corollary 1
If $g(t)$ is a polynomial such that

1. *$g(t)$ is identically equal to 0 or to 1, or*
2. *$g(t)$ is strictly greater than 0 and less than 1, for $0 < t < 1$, and both $g(0)$ and $g(1)$ are greater than or equal to 0 and less than or equal to 1, i.e., $0 < g(t) < 1, \forall t \in (0,1)$ and $0 \le g(0), g(1) \le 1$,*

then the computation of this polynomial can be implemented by stochastic logic. □

This condition is actually *necessary* as well. Since the input argument t and the polynomial evaluation $g(t)$ correspond to probability values in stochastic logic, we require that $0 \le g(t) \le 1$, when $0 \le t \le 1$. Moreover, it can be shown that if $g(t)$ is not identically equal to 0 or to 1 and there exists a $0 < t' < 1$ such that $g(t') = 0$ or 1, then we *cannot* build stochastic logic to compute the polynomial.

If we are given a power-form polynomial $g(t) = \sum_{i=0}^{n} a_i^n t^i$, which satisfies the condition given in Corollary 1, then we can synthesize stochastic logic to compute the polynomial in the following steps:

1. Let $m = n$. Get $b_0^m, b_1^m, \ldots, b_m^m$ from $a_0^n, a_1^n, \ldots, a_n^n$ by Equation (5).
2. Check to see if $0 \le b_i^m \le 1$, for all $i = 0, 1, \ldots, m$. If so, go to step 4.
3. Let $m = m + 1$. Calculate $b_0^m, b_1^m, \ldots, b_m^m$ from $b_0^{m-1}, b_1^{m-1}, \ldots, b_{m-1}^{m-1}$ based on Equation (6). Go to step 2.
4. Build the stochastic logic shown in Section 3.2 to implement the Bernstein polynomial $B^m(t) = \sum_{i=0}^{m} b_i^m B_i^m(t)$.

4. EXPERIMENTAL RESULTS

In our experiments, we first compare the hardware cost of deterministic digital implementations to that of stochastic implementations. Then, we compare the performance of these two implementations on noisy input data.

4.1 Hardware Comparison

In a deterministic implementation of polynomial arithmetic, the data is generally encoded as a binary radix. We assume that the data consists of M bits, so the resolution of the computation is 2^{-M}.

A polynomial $g(t) = \sum_{i=0}^{n} a_i^n t^i$ can be factorized as $g(t) = a_0^n + t(a_1^n + t(a_2^n + \cdots + t(a_{n-1}^n + ta_n^n)))$. With such a factorization, we can evaluate the polynomial in n iterations. In each iteration, a single addition and a single multiplication are needed. Hence, for such an iterative calculation, the hardware consists of an adder and a multiplier.

We build the M-bit multiplier based on the logic design of the ISCAS'85 circuit C6288, given in the benchmark as 16 bits [11]. The C6288 circuit is typical of the genre, built with *carry-save adders*. It consists of 240 full- and half-adder cells arranged in a 15×16 matrix. Each full adder is realized by 9 NOR gates. Incorporating the M-bit adder into the adder matrix of the M-bit multiplier and optimizing it, the circuit requires $10M^2 - 4M - 9$ gates; these are inverters, fanin-2 AND gates, fanin-2 OR gates, and fanin-2 NOR gates. The critical path of the circuit passes through $12M - 11$ logic gates.

We build the implementation computing the Bernstein polynomial of degree n based on the circuit structure shown in Figure 3. Table 1 shows the area $A(n)$ and delay $D(n)$ of our stochastic implementation for Bernstein polynomials of degree $n = 3, 4, 5$, and 6. Each circuit is composed of the same four types of gates that we used in the deterministic implementation. For each specific value of n, we also properly designed the adder tree shown in Figure 4. For example, for $n = 3$, we used a full adder to construct the adder tree, since a full adder takes 3 inputs and gives a 2-bit sum. When characterizing the area and delay, we assumed that the operation of each logic gate requires unit area and unit delay.

Table 1: The area and delay of a single copy of stochastic logic computing Bernstein polynomials of degree 3, 4, 5, and 6.

degree n of Bern. poly.	area $A(n)$	delay $D(n)$
3	22	10
4	40	17
5	49	20
6	58	20

As stated in Section 2.2, the result of the stochastic computation is obtained as the fractional weight of the 1's in the output bit stream. Hence, the resolution of the computation by a bit stream of N bits is $1/N$. Thus, in order to get the same resolution as the deterministic implementation, we need $N = 2^M$. Therefore, we need 2^M *cycles* to get the result when using a serial implementation; alternatively, we need 2^M *copies* when using a parallel implementation.

As a measure of hardware cost, we compute the area-delay product. Note that a serial implementation of stochastic logic takes less area and more delay; a parallel implementation takes more area and less delay. In both cases, the area-delay product is the same.

The area-delay product of the deterministic implementation computing a polynomial of degree n is $(10M^2 - 4M - 9)(12M - 11)n$, where n accounts for the n iterations in the implementation. The area-delay product of the stochastic implementation computing the Bernstein polynomial of degree n is $A(n)D(n)2^M$ – no matter whether implemented serially or in parallel – where $A(n)$ and $D(n)$ are the area and delay of a single copy of stochastic logic, respectively.

In Table 2, we compare the area-delay product for the deterministic implementation and the stochastic implementation for $n = 3, 4, 5, 6$ and $M = 7, 8, 9, 10, 11$. The last column of the table shows the ratio of the area-delay product of the stochastic implementation to that of the deterministic implementation. We can see that when $M \leq 8$, the area-delay product of the stochastic implementation is less than that of the deterministic implementation and when $M \leq 10$, the area-delay product of the stochastic implementation is less than twice that of the deterministic implementation.

Table 2: The area-delay product comparison of the deterministic implementation and the stochastic implementation of polynomials with different degree n and resolution 2^{-M}.

n	M	area-delay product		stoch. prod. / deter. prod.
		deter. impl.	stoch. impl.	
3	7	99207	28160	0.284
	8	152745	56320	0.369
	9	222615	112640	0.506
	10	310977	225280	0.724
	11	419991	450560	1.073
4	7	132276	87040	0.658
	8	203660	174080	0.855
	9	296820	348160	1.173
	10	414636	696320	1.679
	11	559988	1392640	2.487
5	7	165345	125440	0.759
	8	254575	250880	0.986
	9	371025	501760	1.352
	10	518295	1003520	1.936
	11	699985	2007040	2.867
6	7	198414	148480	0.748
	8	305490	296960	0.972
	9	445230	593920	1.334
	10	621954	1187840	1.910
	11	839982	2375680	2.828

4.2 Comparison of Circuit Performance on Noisy Input Data

We compare the performance of deterministic vs. stochastic computation on polynomial evaluations when the input data is corrupted with noise. 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 ϵ 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.

To measure the impact of the noise, we performed two sets of experiments. In the first, we chose the 6-th order Maclaurin polynomial approximation of 11 elementary functions as our implementation target. We list these 11 functions in Table 3, together with the degree of their 6-th order Maclaurin polynomials. Such Maclaurin approximations are commonly used in numerical evaluation of non-polynomial functions.

All of these 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 maximal value; this is a necessary condition for computation with stochastic logic. The scaling factors that we used are listed in Table 3.

We evaluated each Maclaurin polynomial on 13 points: $0.2, 0.25, 0.3, \ldots, 0.8$. For each error ratio ϵ, each Maclaurin

Table 3: Sixth-order Maclaurin polynomial approximation of elementary functions.

function	degree of Mac. poly.	scaling factor
$\sin(x)$	5	N/A
$\tan(x)$	5	0.6818
$\arcsin(x)$	5	0.8054
$\arctan(x)$	5	N/A
$\sinh(x)$	5	0.8511
$\tanh(x)$	5	N/A
$\text{arcsinh}(x)$	5	N/A
$\cos(x)$	6	N/A
$\cosh(x)$	6	0.6481
$\exp(x)$	6	0.3679
$\ln(x+1)$	6	N/A

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 ϵ, we averaged the relative errors over all polynomials and all evaluation points.

In the second set of experiments, we randomly choose 100 Bernstein polynomials of degree 6 with coefficients in the unit interval. With this specification, we are guaranteed that the computation can be implemented using stochastic logic. We evaluated each on 10 points: $0, 1/9, 2/9, \ldots, 1$. We compiled similar statistics to that in the first set of experiments. Table 4 shows the average relative error of the stochastic implementation and the deterministic implementation versus different error ratios ϵ for both sets of experiments. We plot the data for the experiments on Maclaurin polynomials in Figure 5 to give a clear comparison.

Table 4: Relative error for the stochastic implementation and deterministic implementation of polynomial computation versus the error ratio ϵ in the input data.

	Maclaurin poly.		Randomly chosen poly.	
error ratio ϵ	rel. error of stoch. impl.(%)	rel. error of deter. impl.(%)	rel. error of stoch. impl.(%)	rel. error of deter. impl.(%)
0.0	2.63	0.00	2.92	0.00
0.001	2.62	0.68	3.06	11.1
0.002	2.64	1.41	3.27	21.3
0.005	2.73	3.36	4.25	53.9
0.01	3.01	6.75	6.05	106
0.02	3.89	12.8	9.93	208
0.05	7.54	28.9	21.4	494
0.1	13.8	51.2	39.2	948

When $\epsilon = 0$, meaning that no noise is injected into the input data, the deterministic implementation computes without any error. However, due to the inherent variance, the stochastic implementation produces a small relative error. However, with noise, the relative error of the deterministic implementation blows up dramatically as ϵ increases. Even for small values, the stochastic implementation performs much better.

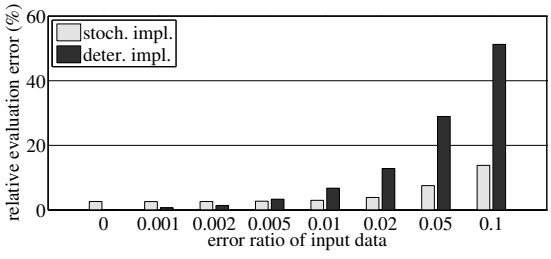

Figure 5: A plot of the relative error for the stochastic and the deterministic implementation of Maclaurin polynomial computation versus the error ratio ϵ in the input data.

In Table 4, note that the relative evaluation error of the randomly chosen polynomials computed by the deterministic implementation is much larger than that of the Maclaurin polynomials. The explanation for this is that the randomly chosen polynomials have much larger power-form coefficients than the Maclaurin polynomials do. Thus, bit flips on the coefficients dramatically change their evaluation.

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.

5. CONCLUSION AND FUTURE WORK

The synthesis results for the stochastic implementation of polynomial arithmetic are convincing. The area-delay product is comparable to that of deterministic implementations with adders and multipliers. However, the circuits are much more error-tolerant. The precision of the results is dependent only on the statistics of the streams that flow through the datapaths, and so the computation can tolerate errors gracefully.

In this paper, we focused on noisy input data, assuming that the logic itself is fault free. For emerging technologies, such as self-assembled nanowire circuits, this assumption must be challenged: the logic and the interconnects may also be unreliable [4], In future work, we will extend our methodology to stochastic computation of arithmetic functions with probabilistic logic and interconnects as well as signal values.

6. REFERENCES

[1] Z. Asgar, S. Kodakara and D. Lilja, "Fault-tolerant image processing using stochastic logic," *Technical Report*, http://www.zasgar.net/zain/publications/publications.php, 2005.

[2] C.M. Bishop, *Neural networks for patten recognition*, Clarendon Press, 1995.

[3] B. Brown and H. Card, "Stochastic neural computation I: computational elements," *IEEE Transactions on Computers*, Vol. 50, No. 9, pp. 891–905, 2001.

[4] A. DeHon, "Nanowire-based programmable architectures," *ACM Journal on Emerging Technologies in Computing Systems*, Vol. 1, No. 2, pp. 109–162, 2005.

[5] S.R. Deiss, R.J. Douglas and A.M. Whatley, "A pulse coded communications infrastructure for neuromorphic systems," *Pulsed Neural Networks*, W. Maass and C.M. Bishop, eds., Chapter 6, MIT Press, 1999.

[6] R.T. Farouki and V.T. Rajan, "On the numerical condition of polynomials in Bernstein form," *Computer-Aided Geometric Design*, Vol. 4, No. 3, pp. 191–216, 1987.

[7] B.R. Gaines, "Stochastic computing systems," *Advances in Information Systems Science*, J.F. Tou, ed., Vol. 2, Chapter 2, pp. 37–172, Plenum, 1969.

[8] C.L. Janer, J.M. Quero, J.G. Ortega and L.G. Franquelo, "Fully parallel stochastic computation architecture," *IEEE Transactions on Signal Processing*, Vol. 44, No. 8, pp. 2110–2117, 1996.

[9] G. Lorentz, *Bernstein polynomials,* Univ. of Toronto Press, 1953.

[10] S.L. Toral, J.M. Quero and L.G. Franquelo, "Stochastic pulse coded arithmetic," *Proceedings of ISCAS*, pp. 599–602, 2000.

[11] "ISCAS'85 C6288 16x16 multiplier" http://www.eecs.umich.edu/∼jhayes/iscas/c6288.html