An Architecture for Fault-Tolerant Computation with Stochastic Logic

Weikang Qian, Xin Li, Marc D. Riedel, Kia Bazargan, and David J. Lilja

Abstract—Mounting concerns over variability, defects and noise motivate a new approach for digital circuitry: stochastic logic, that is to say, logic that operates on probabilistic signals and so can cope with errors and uncertainty. Techniques for probabilistic *analysis* of circuits and systems are well established. We advocate a strategy for *synthesis*. In prior work, we described a methodology for synthesizing stochastic logic, that is to say logic that operates on probabilistic bit streams. In this paper, we apply the concept of stochastic logic to a reconfigurable architecture that implements processing operations on a datapath. We analyze cost as well as the sources of error: approximation, quantization, and random fluctuations. We study the effectiveness of the architecture on a collection of benchmarks for image processing. The stochastic architecture requires less area than conventional hardware implementations. Moreover, it is much more tolerant of soft errors (bit flips) than these deterministic implementations. This fault tolerance scales gracefully to very large numbers of errors.

Index Terms—Stochastic Logic, Reconfigurable Hardware, Fault-Tolerant Computation

1 Introduction

The successful design paradigm for integrated circuits has been rigidly hierarchical, with sharp boundaries between different levels of abstraction. From the logic level up, the precise Boolean functionality of the system is fixed and deterministic. This abstraction is costly: variability and uncertainty at the circuit level must be compensated for by better physical design. With increased scaling of semiconductor devices, soft errors caused by ionizing radiation are a major concern, particularly for circuits operating in harsh environments such as space. Existing methods mitigate against bit-flips with system-level techniques like error-correcting codes and modular redundancy.

Randomness, however, is a valuable resource in computation. A broad class of algorithms in areas such as cryptography and communication can be formulated with lower complexity if physical sources of randomness are available [1], [2]. Applications that entail the simulation of random physical phenomena, such as computational biology and quantum physics, also hinge upon randomness (or good pseudo-randomness) [3].

We advocate a novel view for computation, called *stochastic logic*. Instead of designing circuits that transform definite inputs into definite outputs – say Boolean, integer, or floating-point values into the same – we synthesize circuits that conceptually transform probability values into probability values. The approach is applicable for randomized algorithms. It is also applicable for data intensive applications such as signal processing where small fluctuations can be tolerated but large errors are catastrophic. In such contexts, our approach offers savings in computational resources and provides significantly better fault tolerance.

- *This work is supported by a grant from the Semiconductor Research Corporation's Focus Center Research Program on Functional Engineered Nano-Architectonics, contract No. 2003-NT-1107, a CAREER Award, #0845650, from the National Science Foundation, and a grant from Intel Corporation.*
- *The authors are with the Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN, 55455, USA. E-mail: {qianx030,lixxx914,mriedel,kia,lilja}@umn.edu*

1.1 Stochastic Logic

In prior work, we described a methodology for synthesizing stochastic logic [4]. Operations at the logic level are performed on randomized values in serial streams or on parallel "bundles" of wires. When serially streaming, the signals are probabilistic in *time*, as illustrated in Figure 1(a); in parallel, they are probabilistic in *space*, as illustrated in Figure 1(b).

The bit streams or wire bundles are digital, carrying zeros and ones; they are processed by ordinary logic gates, such as AND and OR. However, the signal is conveyed through the statistical distribution of the logical values. With physical uncertainty, the fractional numbers correspond to the probability of occurrence of a logical one versus a logical zero. In this way, computations in the deterministic Boolean domain are transformed into probabilistic computations in the real domain. In the serial representation, a real number x in the unit interval (i.e., $0 \leq x \leq 1$) corresponds to a bit stream $X(t)$ of length N, $t = 1, 2, \ldots, N$. In the parallel representation, it corresponds to the bits on a bundle of N wires. The probability that each bit in the stream or the bundle is one is $P(X = 1) = x$.

Throughout this paper, we illustrate our method with serial bit streams. However, our approach is equally applicable to parallel wire bundles. Indeed, we have advocated stochastic logic as a framework for synthesis for technologies such as nanowire crossbar arrays [5].

Fig. 1: Stochastic encoding: (a) A stochastic bit stream; (b) A stochastic wire bundle. A real value x in $[0, 1]$ is represented as a bit stream or a bundle X. For each bit in the bit stream or bundle, the probability that it is 1 is $P(X = 1) = x$.

Our synthesis strategy is to cast logical computations as arithmetic operations in the probabilistic domain and implement these directly as stochastic operations on data-paths. Two simple arithmetic operations – multiplication and scaled addition – are illustrated in Figure 2.

- **Multiplication**. Consider a two-input AND gate, shown in Figure 2(a). Suppose that its inputs are two independent bit streams X_1 and X_2. Its output is a bit stream Y, where

$$\begin{aligned} y &= P(Y = 1) = P(X_1 = 1 \text{ and } X_2 = 1) \\ &= P(X_1 = 1)P(X_2 = 1) = x_1 x_2. \end{aligned}$$

  Thus, the AND gate computes the product of the two input probability values.
- **Scaled Addition**. Consider a two-input multiplexer, shown in Figure 2(b). Suppose that its inputs are two independent stochastic bit streams X_1 and X_2 and its selecting input is a stochastic bit stream S. Its output is a bit stream Y, where

$$\begin{aligned} y &= P(Y = 1) \\ &= P(S = 1)P(X_1 = 1) + P(S = 0)P(X_2 = 1) \\ &= sx_1 + (1 - s)x_2. \end{aligned}$$

  (Note that throughout the paper, multiplication and addition represent *arithmetic* operations, not Boolean AND and OR.) Thus, the multiplexer computes the scaled addition of the two input probability values.

More complex functions such as division, the Taylor expansion of the exponential function, and the square root function can also be implemented with only a dozen or so gates each using the stochastic methodology. Prior work established specific constructs for such operations [6]–[8]. We tackle the problem more broadly: we propose a synthesis methodology for stochastic computation.

The stochastic approach offers the advantage that complex operations can be performed with very simple logic. Of course the method entails redundancy in the encoding of signal values. Signal values are fractional values corresponding to the probability of logical one. If the resolution of a computation is required to be 2^{-M} then the length or width of the bit stream should be 2^M bits. This is a significant trade-off in time (for a serial encoding) or in space (for a parallel encoding).

1.2 Fault Tolerance

The advantage of the stochastic architecture in terms of resources is that it tolerates faults gracefully. Compare a stochastic encoding to a standard binary radix encoding, say with M bits representing fractional values between 0 and 1. Suppose that the environment is noisy; bit flips occur and these afflict all the bits with equal probability. With a binary radix encoding, suppose that the most significant bit of the data gets flipped. This causes a relative error of $2^{M-1}/2^M = 1/2$. In contrast, with a stochastic encoding, 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 small in comparison.

Figure 3 illustrates the fault tolerance that our approach provides. The circuit in Figure 3(a) is a stochastic implementation while the circuit in Figure 3(b) is a conventional implementation. Both circuits compute the function:

$$y = x_1 x_2 s + x_3(1 - s).$$

Consider the stochastic implementation. Suppose that the inputs are $x_1 = 4/8$, $x_2 = 6/8$, $x_3 = 7/8$, and $s = 2/8$. The corresponding bit streams are shown above the wires. Suppose that the environment is noisy and bit flips occur at a rate of 10%; this will result in approximately three bit flips for the stream lengths shown. A random choice of three bit flips is shown in the figure. The modified streams are shown below the wires. With these bit flips, the output value changes but by a relatively small amount: from $6/8$ to $5/8$.

In contrast, Figure 3(b) shows a conventional implementation of the function with multiplication and addition modules operating on a binary radix representation: the real numbers $x_1 = 4/8$, $x_2 = 6/8$, $x_3 = 7/8$, and $s = 2/8$ are encoded as $(0.100)_2$, $(0.110)_2$, $(0.111)_2$, and $(0.010)_2$, respectively. The correct result is $y = (0.110)_2$, which equals $6/8$. In the same situation as above, with a 10% rate of bit flips, approximately one bit will get flipped. Suppose that, unfortunately, this is the most significant bit of x_3. As a result, x_3 changes to $(0.011)_2 = 3/8$ and the output y becomes $(0.011_2) = 3/8$. This is a much larger error than we expect with the stochastic implementation.

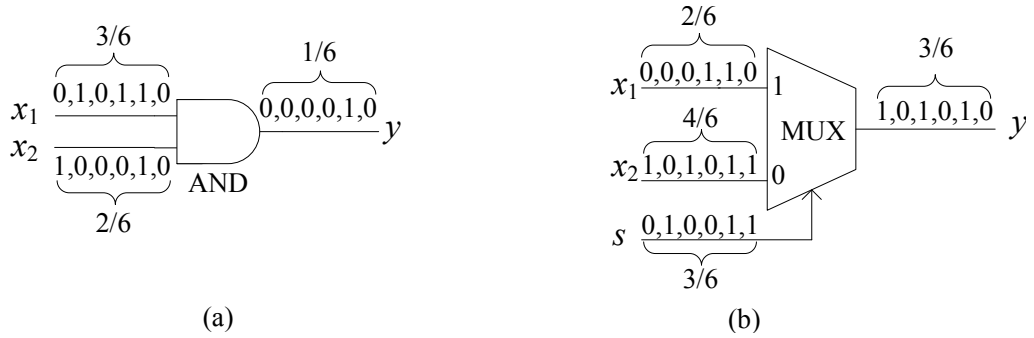
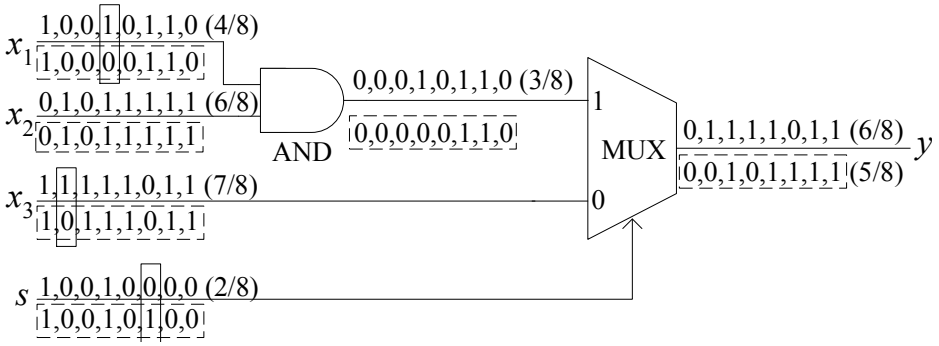
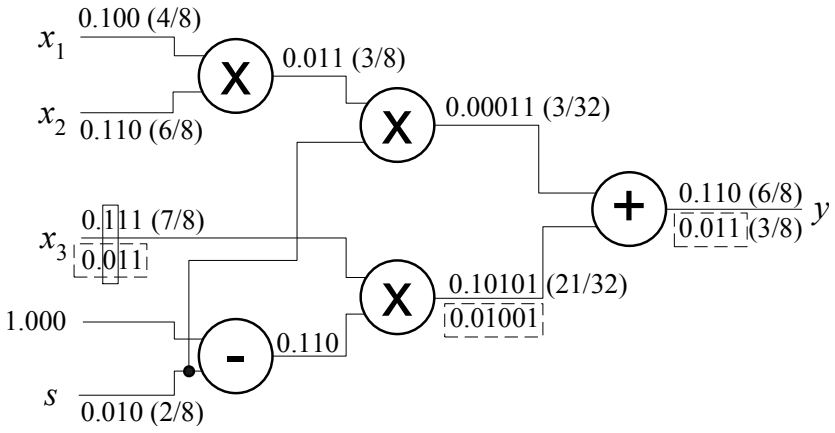


Fig. 2: Stochastic implementation of arithmetic operations: (a) Multiplication; (b) Scaled addition.



(a) Stochastic implementation of the function $y = x_1x_2s + x_3(1-s)$.



(b) Conventional implementation of the function $y = x_1x_2s + x_3(1-s)$, using binary radix multiplier, adder and subtractor units.

Fig. 3: A comparison of the fault tolerance of stochastic logic to conventional logic. The original bit sequence is shown above each wire. A bit flip is indicated with a solid rectangle. The modified bit sequence resulting from the bit flip is shown below each wire and indicated with a dotted rectangle.

1.3 Related Work and Context

The topic of computing reliably with unreliable components dates back to von Neumann and Shannon [9], [10]. Techniques such as modular redundancy and majority voting are widely used for fault tolerance. Error correcting codes are applied for memory subsystems and communication links, both on-chip and off-chip.

Probabilistic methods are ubiquitous in circuit and system design. Generally, they are applied with the aim of characterizing uncertainty. For instance, statistical timing analysis is used to obtain tighter performance bounds [11] and also applied in transistor sizing to maximize yield [12]. . Many flavors of probabilistic design have been proposed for integrated circuits. For instance, [13] presents a design methodology based on Markov random fields geared toward nanotechnology; [14] presents a methodology based on probabilistic CMOS, with a focus on energy efficiency.

There has a promising recent effort to design so-called stochastic processors [15]. The strategy in that work is to deliberately under-design the hardware, such that it is allowed to produce errors, and to implement error

tolerance through software mechanisms. As much as possible, the burden of error tolerance is pushed all the way to the application layer. The approach permits aggressive power reduction in the hardware design. It is particularly suitable for high-performance computing applications, such as Monte Carlo simulations, that naturally tolerate errors.

On the one hand, our work is more narrowly circumscribed: we present a specific architectural design for datapath computations. On the other hand, our contribution is a significant departure from existing methods, predicated on a new logic-level synthesis methodology. We design processing modules that compute in terms of statistical distributions. The modules processes serial or parallel streams that are random at the bit level. In the aggregate, the computation is robust and accurate since the results depend only on the statistics not on specific bit values. The computation is "analog" in character, cast in terms of real-valued probabilities, but it is implemented with digital components. The strategy is orthogonal to specific hardware-based methods for error tolerance, such as error-coding of memory subsystems [16]. It is also compatible with application layer and other software-based methods for error tolerance.

In [4], we presented a methodology for synthesizing arbitrary polynomial functions with stochastic logic. We also extended the method to the computation of arbitrary continuous functions through non-polynomial approximations [17]. In [18], we considered the complementary problem of generating probabilistic signals for stochastic computation. We described a method for transforming arbitrary sources of randomness into the requisite probability values, entirely through combinational logic.

1.4 Overview

In this paper, we apply the concept of stochastic logic to a reconfigurable architecture that implements processing operations on a datapath. We analyze cost as well as the sources of error: approximation, quantization, and random fluctuations. We study the effectiveness of the architecture on a collection of benchmarks for image processing. The stochastic architecture requires less area than conventional hardware implementations. Moreover, it is much more tolerant of soft errors (bit flips) than these deterministic implementations. This fault tolerance scales gracefully to very large numbers of errors.

The rest of the paper is structured as follows. Section 2 discusses the synthesis of stochastic logic. Section 3 presents our reconfigurable architecture. Section 4 analyzes the sources of error in stochastic computation. Section 5 describes our implementation of the architecture. Section 6 provides experimental results. Section 7 presents conclusions and future directions of research.

2 Synthesizing Stochastic Logic

2.1 Synthesizing Polynomials

By definition, the computation of polynomial functions entails multiplications and additions. These can be implemented with the stochastic constructs described in Section 1.1. However, the method fails for polynomials with coefficients less than zero or greater than one, e.g., $1.2x - 1.2x^2$, since we cannot represent such coefficients with stochastic bit streams.

In [4], we proposed a method for implementing arbitrary polynomials, including those with coefficients less than zero or greater than one. As long as the polynomial maps values from the unit interval to values in the unit interval, then no matter how large the coefficients are, we can synthesize stochastic logic that implements it. The procedure begins by transforming a power-form polynomial into a Bernstein polynomial [19]. A Bernstein polynomial of degree n is of the form

$$B(x) = \sum_{i=0}^{n} b_i B_{i,n}(x), \tag{1}$$

where each real number b_i is a coefficient, called a Bernstein coefficient, and each $B_{i,n}(x)(i = 0, 1, \ldots, n)$ is a Bernstein basis polynomial of the form

$$B_{i,n}(x) = \binom{n}{i} x^i (1-x)^{n-i}. \tag{2}$$

A power-form polynomial of degree n can be transformed into a Bernstein polynomial of degree no less than n. Moreover, if a power-form polynomial maps the unit interval onto itself, we can convert it into a Bernstein polynomial with coefficients that are all in the unit interval.

A Bernstein polynomial with all coefficients in the unit interval can be implemented stochastically by a generalized multiplexing circuit, shown in Figure 4. The circuit 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. Thus, the output of the multiplexer y is set to be z_i $(0 \leq i \leq n)$, where i equals the binary number computed by the adder; this is the number of ones in the input set $\{x_1, \ldots, x_n\}$.

The stochastic input bit streams are set as follows:

- The inputs $x_1, \ldots, x_n$ are *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 *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 $B(x) = \sum_{i=0}^{n} b_i B_{i,n}(x)$. We discuss generating and interpreting such input and output bit streams in Sections 3.2 and 3.3.

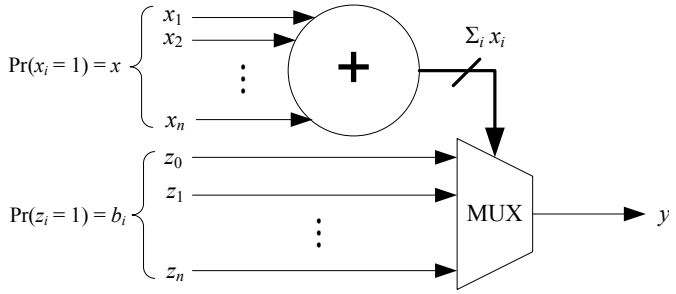


Fig. 4: A generalized multiplexing circuit implementing the Bernstein polynomial $y = B(x) = \sum_{i=0}^{n} b_i B_{i,n}(x)$ with $0 \leq b_i \leq 1$, for $i = 0, 1, \ldots, n$.

Example 1

The polynomial $f_1(x) = \frac{1}{4} + \frac{9}{8}x - \frac{15}{8}x^2 + \frac{5}{4}x^3$ *maps the unit interval onto itself. It can be converted into a Bernstein polynomial of degree* 3*:*

$$f_1(x) = \frac{2}{8}B_{0,3}(x) + \frac{5}{8}B_{1,3}(x) + \frac{3}{8}B_{2,3}(x) + \frac{6}{8}B_{3,3}(x).$$

Notice that all the coefficients are in the unit interval. The stochastic logic that implements this Bernstein polynomial is shown in Figure 5. Assume that the original polynomial is evaluated at $x = 0.5$*. The stochastic bit streams of inputs* x_1*,* x_2 *and* x_3 *are independent and each represents the probability value* $x = 0.5$*. The stochastic bit streams of inputs* $z_0, \ldots, z_3$ *represent probabilities* $b_0 = \frac{2}{8}$*,* $b_1 = \frac{5}{8}$*,* $b_2 = \frac{3}{8}$*, and* $b_3 = \frac{6}{8}$*. As expected, the stochastic logic computes the correct output value:* $f_1(0.5) = 0.5$*.* □

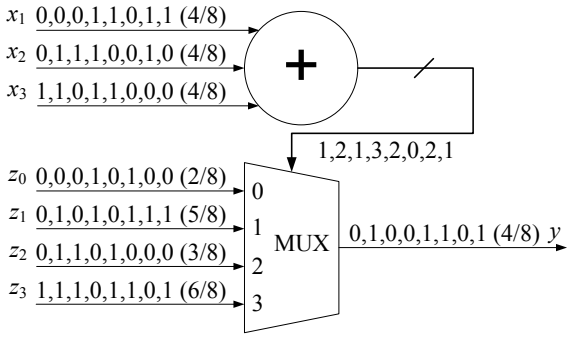


Fig. 5: Stochastic logic implementing the Bernstein polynomial $f_1(x) = \frac{2}{8}B_{0,3}(x) + \frac{5}{8}B_{1,3}(x) + \frac{3}{8}B_{2,3}(x) + \frac{6}{8}B_{3,3}(x)$ at $x = 0.5$. Stochastic bit streams x_1, x_2 and x_3 encode the value $x = 0.5$. Stochastic bit streams z_0, z_1, z_2 and z_3 encode the corresponding Bernstein coefficients.

2.2 Synthesizing Non-Polynomial Functions

It was proved in [4] that stochastic logic can *only* implement polynomial functions. In real applications, of course, we often encounter non-polynomial functions, such as trigonometric functions. A method was proposed in [17] to synthesize arbitrary functions by approximating them via Bernstein polynomial. Indeed, given a continuous function $f(x)$ of degree n as the target, a set of real coefficients $b_0, b_1, \ldots, b_n$ in the interval $[0, 1]$ are sought to minimize the objective function

$$\int_0^1 (f(x) - \sum_{i=0}^{n} b_i B_{i,n}(x))^2 \, \mathrm{d}x, \tag{3}$$

By expanding Equation (3), an equivalent objective function can be obtained:

$$f(\boldsymbol{b}) = \frac{1}{2}\boldsymbol{b}^T \boldsymbol{H} \boldsymbol{b} + \boldsymbol{c}^T \boldsymbol{b}, \tag{4}$$

where

$$\boldsymbol{b} = [b_0, \ldots, b_n]^T,$$

$$\boldsymbol{c} = [-\int_0^1 f(x)B_{0,n}(x)\,\mathrm{d}x, \ldots, -\int_0^1 f(x)B_{n,n}(x)\,\mathrm{d}x]^T,$$

$$\boldsymbol{H} = \begin{bmatrix} \int_0^1 B_{0,n}(x)B_{0,n}(x)\,\mathrm{d}x & \ldots & \int_0^1 B_{0,n}(x)B_{n,n}(x)\,\mathrm{d}x \\ \int_0^1 B_{1,n}(x)B_{0,n}(x)\,\mathrm{d}x & \ldots & \int_0^1 B_{1,n}(x)B_{n,n}(x)\,\mathrm{d}x \\ \vdots & \ddots & \vdots \\ \int_0^1 B_{n,n}(x)B_{0,n}(x)\,\mathrm{d}x & \ldots & \int_0^1 B_{n,n}(x)B_{n,n}(x)\,\mathrm{d}x \end{bmatrix}$$

This optimization problem is, in fact, a constrained quadratic programming problem. Its solution can be obtained using standard techniques. Once we obtain the requisite Bernstein coefficients, we can implement the polynomial approximation as a Bernstein computation with the generalized multiplexing circuit described in Section 2.1.

Example 2

Gamma Correction. *The gamma correction function is a nonlinear operation used to code and decode luminance and tri-stimulus values in video and still-image systems. It is defined by a power-law expression*

$$V_{out} = V_{in}^{\gamma},$$

where V_{in} *is normalized between zero and one [20]. We apply a value of* $\gamma = 0.45$*, which is the value used in most TV cameras.*

Consider the non-polynomial function

$$f_2(x) = x^{0.45}.$$

We approximate this function by a Bernstein polynomial of degree 6. By solving the constrained quadratic optimization problem, we obtain the Bernstein coefficients:

$$b_0 = 0.0955, b_1 = 0.7207, b_2 = 0.3476, b_3 = 0.9988,$$
$$b_4 = 0.7017, b_5 = 0.9695, b_6 = 0.9939. \qquad \square$$

In a strict mathematical sense, stochastic logic can only implement functions that map the unit interval into the unit interval. However, with scaling, stochastic logic can implement functions that map any *finite* interval into any *finite* interval. For example, the functions used in grayscale image processing are defined on the interval $[0, 255]$ with the same output range. If we want to implement such a function $y = f(t)$, we can instead implement the function $y = g(t) = \frac{1}{256}f(256t)$. Note that the new function $g(t)$ is defined on the unit interval and its output is also on the unit interval.

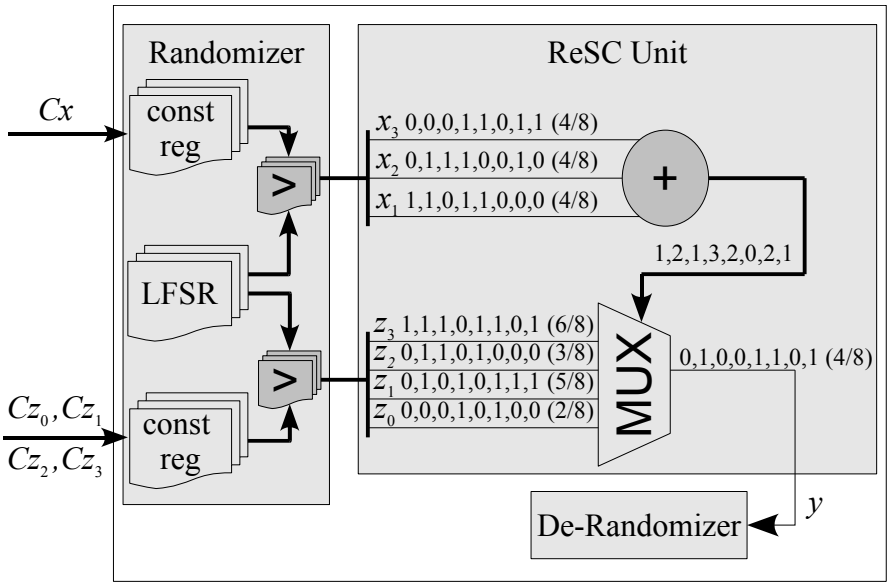


Fig. 6: A reconfigurable stochastic computing architecture. Here the ReSC Unit implements the target function $y = \frac{1}{4} + \frac{9}{8}x - \frac{15}{8}x^2 + \frac{5}{4}x^3$ at $x = 0.5$.

3 The Stochastic Architecture

We present a novel **Re**configurable architecture based on **S**tochastic logi**C**: the **ReSC architecture**. As illustrated in Figure 6, it composed of three parts: the *Randomizer Unit* generates stochastic bit streams; the *ReSC Unit* processes these bit streams; and the *De-Randomizer Unit* converts the resulting bit streams to output values. The architecture is reconfigurable in the sense that it can be used to compute different functions by setting appropriate values of *constant registers*.

3.1 The ReSC Unit

The ReSC Unit is the kernel of the architecture. It is the generalized multiplexing circuit described in Section 2.1, which implements Bernstein polynomials with coefficients in the unit interval. As described in Section 2.2, we can use it to approximate arbitrary continuous functions.

The probability x of the independent stochastic bit streams x_i is controlled by the binary number C_x in a constant register, as illustrated in Figure 6. The constant register is a part of the Randomizer Unit, discussed below. Similarly, stochastic bit streams $z_0, \ldots, z_n$ representing a specified set of coefficients can be produced by configuring the binary numbers C_{z_i}'s in constant registers.

3.2 The Randomizer Unit

The Randomizer Unit is shown in Figure 7. To generate a stochastic bit stream, a random number generator produces a number R in each clock cycle. If R is strictly less than the number C stored in the corresponding constant number register, then the comparator generates a one; otherwise, it generates a zero.

In our implementation, we use linear feedback shift registers (LFSRs). Assume that an LFSR has L bits.

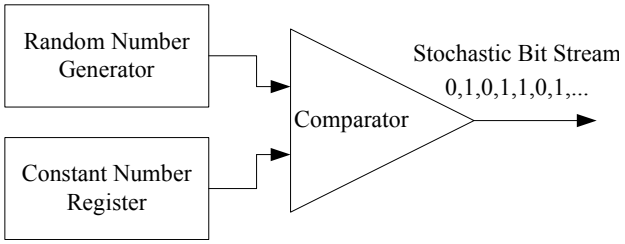


Fig. 7: The Randomizer Unit.

Accordingly, it generates repeating pseudorandom numbers with a period of $2^L - 1$. We choose L so that $2^L - 1 \geq N$, where N is the length of the input random bit streams. This guarantees good randomness of the input bit streams. The set of random numbers that can be generated by such an LFSR is $\{1, 2, \ldots, 2^L - 1\}$ and the probability that R equals a specific k in the set is

$$P(R = k) = \frac{1}{2^L - 1}. \tag{5}$$

Given a constant integer $1 \leq C \leq 2^L$, the comparator generates a one with probability

$$P(R < C) = \sum_{k=1}^{C-1} P(R = k) = \frac{C - 1}{2^L - 1}. \tag{6}$$

Thus, the set of probability values that can be generated by the Randomizer Unit is

$$S = \{0, \frac{1}{2^L - 1}, \ldots, 1\}. \tag{7}$$

Given an arbitrary value $0 \leq p \leq 1$, we round it to the closest number p' in S. Hence, C is determined by p as

$$C = \text{round}(p(2^L - 1)) + 1, \tag{8}$$

where the function round(x) equals the integer nearest to x. The value p is quantized to

$$p' = \frac{\text{round}(p(2^L - 1))}{2^L - 1} \tag{9}$$

In our stochastic implementation, we require different input random bit streams to be independent. Therefore, LFSRs for generating different input random bit streams are configured to have different feedback functions.

3.3 The De-Randomizer Unit

The De-Randomizer Unit translates the result of the stochastic computation, expressed as a randomized bit stream, back to a deterministic value using a counter. We set the length of the stochastic bit stream to be a power of 2, i.e., $N = 2^M$, where M is an integer. We choose the number of bits of the counter to be $M + 1$, so that we can count all possible numbers of ones in the stream: from 0 to 2^M ones. We treat the output of the counter as a binary decimal number $d = (c_M.c_{M-1} \dots c_0)_2$, where $c_0, c_1, \dots, c_M$ are the $M + 1$ output bits of the counter.

Since each bit of the stream X has probability x of being one, the mean value of the counter result d is

$$\begin{aligned} E[d] &= E\left[\frac{(c_M \dots c_0)_2}{2^M}\right] = E\left[\frac{1}{N}\sum_{\tau=1}^{N} X(\tau)\right] \\ &= \frac{1}{N}\sum_{\tau=1}^{N} E[X(\tau)] = x, \end{aligned} \tag{10}$$

which is the value represented by the bit stream X.

Compared to the kernel, the Randomizer and De-Randomizer units are expensive in terms of the hardware resources required. Indeed, they dominate the area cost of the architecture. We note that in some applications, both the Randomizer and De-Randomizer units could be implemented directly through physical interfaces. 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 [21]. Also, random bit streams with specific probabilities can be generated from fixed sources of randomness through combinational logic. In [18], we presented a method for synthesizing logic that generates arbitrary sets of output probabilities from a small given set of input probabilities.

4 The Error Analysis

By its very nature, stochastic logic introduces uncertainty into the computation. There are three sources of errors.

1) **The error due to the Bernstein approximation**: Since we use a Bernstein polynomial with coefficients in the unit interval to approximate a function $g(t)$, there is some approximation error

$$e_1 = \left| g(t) - \sum_{i=0}^{n} b_{i,n} B_{i,n}(t) \right|. \tag{11}$$

We could use the L^2-norm to measure the average error as

$$\begin{aligned} e_{1\text{avg}} &= \left(\frac{1}{1-0} \int_0^1 (g(t) - \sum_{i=0}^{n} b_{i,n} B_{i,n}(t))^2 \, \mathrm{d}t \right)^{0.5} \\ &= \left(\int_0^1 (g(t) - \sum_{i=0}^{n} b_{i,n} B_{i,n}(t))^2 \, \mathrm{d}t \right)^{0.5} \end{aligned} \tag{12}$$

The average approximation error e_{1avg} only depends on the original function $g(t)$ and the degree of the Bernstein polynomial; e_{1avg} decreases as n increases. For all of the functions that we tested, a Bernstein approximation of degree of 6 was sufficient to reduce e_{1avg} to below 10^{-3}.†

2) **The quantization error**:
As shown in Section 3.2, given an arbitrary value $0 \le p \le 1$, we round it to the closest number p' in $S = \{0, \frac{1}{2^L-1}, \dots, 1\}$ and generate the corresponding bit stream. Thus, the quantization error for p is

$$|p - p'| \le \frac{1}{2(2^L - 1)}, \tag{13}$$

where L is the number of bits of the LFSR.

Due to the effect of quantization, we will compute $\sum_{i=0}^{n} b'_{i,n} B_{i,n}(t')$ instead of the Bernstein approximation $\sum_{i=0}^{n} b_{i,n} B_{i,n}(t)$, where $b'_{i,n}$ and t' are the closest numbers to $b_{i,n}$ and t, respectively, in set S. Thus, the quantization error is

$$e_2 = \left| \sum_{i=0}^{n} b'_{i,n} B_{i,n}(t') - \sum_{i=0}^{n} b_{i,n} B_{i,n}(t) \right| \tag{14}$$

Define $\Delta b_{i,n} = b'_{i,n} - b_{i,n}$ and $\Delta t = t' - t$. Then, using a first order approximation, the error due to quantization is

$$\begin{aligned} e_2 &\approx \left| \sum_{i=0}^{n} B_{i,n}(t) \Delta b_{i,n} + \sum_{i=0}^{n} b_{i,n} \frac{\mathrm{d}B_{i,n}(t)}{\mathrm{d}t} \Delta t \right| \\ &= \left| \sum_{i=0}^{n} B_{i,n}(t) \Delta b_{i,n} + n \sum_{i=0}^{n-1} (b_{i+1,n} - b_{i,n}) B_{i,n-1}(t) \Delta t \right| \end{aligned}$$

Notice that since $0 \le b_{i,n} \le 1$, we have $|b_{i+1,n} - b_{i,n}| \le 1$. Combining this with the fact that $\sum_{i=0}^{n} B_{i,n}(t) = 1$ and $|\Delta b_{i,n}|, |\Delta t| \le \frac{1}{2(2^L-1)}$,

†. For many applications, 10^{-3} would be considered a low error rate. As discussed in Section 6, due to inherent stochastic variation, our stochastic implementation has larger output errors than conventional implementations when the input error rate is low. Thus, our system targets noisy environments with relatively high input error rates – generally, larger than 0.01.

we have

$$e_2 \le \frac{1}{2(2^L-1)}\left|\sum_{i=0}^{n} B_{i,n}(t)\right| + \frac{n}{2(2^L-1)}\left|\sum_{i=0}^{n-1} B_{i,n-1}(t)\right| = \frac{n+1}{2(2^L-1)}. \quad (15)$$

Thus, the quantization error is inversely proportional to 2^L. We can mitigate this error by increasing the number of bits L of the LFSR.

3) **The error due to random fluctuations**: Due to the Bernstein approximation and the quantization effect, the output bit stream $Y(\tau)$ $(\tau = 1, 2, \ldots, N)$ has probability $p' = \sum_{i=0}^{n} b'_{i,n} B_{i,n}(t')$ that each bit is one. The De-Randomizer Unit returns the result

$$y = \frac{1}{N}\sum_{\tau=1}^{N} Y(\tau). \quad (16)$$

It is easily seen that $E[y] = p'$. However, the realization of y is not, in general, exactly equal to p'. The error can be measured by the variation as

$$Var[y] = Var[\frac{1}{N}\sum_{\tau=1}^{N} Y(\tau)] = \frac{1}{N^2}\sum_{\tau=1}^{N} Var[Y(\tau)] = \frac{p'(1-p')}{N}. \quad (17)$$

Since $Var[y] = E[(y - E[y])^2] = E[(y - p')^2]$, the error due to random fluctuations is

$$e_3 = |y - p'| \approx \sqrt{\frac{p'(1-p')}{N}}. \quad (18)$$

Thus, the error due to random fluctuations is inversely proportional to $\sqrt{N}$. Increasing the length of the bit stream will decrease the error.

The overall error is bounded by the sum of the above three error components:

$$e = |g(t) - y| \le \left|g(t) - \sum_{i=0}^{n} b_{i,n} B_{i,n}(t)\right| + \left|\sum_{i=0}^{n} b_{i,n} B_{i,n}(t) - \sum_{i=0}^{n} b'_{i,n} B_{i,n}(t')\right| + \left|\sum_{i=0}^{n} b'_{i,n} B_{i,n}(t') - y\right| = e_1 + e_2 + e_3. \quad (19)$$

Note that we choose the number of bits L of the LFSRs to satisfy $2^L - 1 \ge N$ in order to get non-repeating random bit streams. Therefore, we have

$$\frac{1}{2^L} < \frac{1}{N} \ll \frac{1}{\sqrt{N}}$$

Combining the above equation with Equations (15) and (18), we can see that in our implementation, the error due to random fluctuations will dominate the quantization error. Therefore, the overall error e is approximately bounded by the sum of error e_1 and e_3, i.e.,

$$e \le e_1 + e_3.$$

5 Implementation

The top-level block diagram of the system is illustrated in Figure 8. A MicroBlaze 32-bit soft RISC processor core is used as the main processor. Our ReSC is configured as a coprocessor, handled by the Microblaze. The MicroBlaze talks to the ReSC unit through a Fast Simplex Link (FSL), a FIFO-style connection bus system [22]. (The MicroBlaze is the master; the ReSC unit is the slave.)

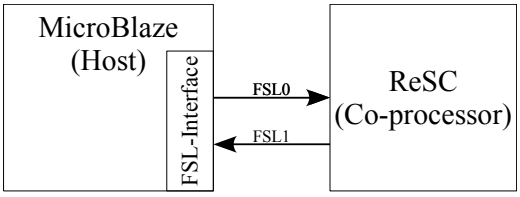


Fig. 8: Overview of the architecture of the ReSC system.

Consider the example of the gamma function, discussed in Example 2. We approximate this function by a Bernstein polynomial of degree 6 with coefficients:

$$b_0 = 0.0955, b_1 = 0.7207, b_2 = 0.3476, b_3 = 0.9988,$$
$$b_4 = 0.7017, b_5 = 0.9695, b_6 = 0.9939$$

In our implementation, the LFSR has 10 bits. Thus, by (8), the numbers that we load into in the constant coefficient registers are:

$$C_0 = 99, C_1 = 738, C_2 = 357, C_3 = 1023,$$
$$C_4 = 719, C_5 = 993, C_6 = 1018.$$

Figure 9 illustrates how we specify C code to implements the gamma correction function on the ReSC architecture. Such code is compiled by the MicroBlaze C compiler. The coefficients for the Bernstein computation are specified in lines 4 to 7. These are loaded into registers in lines 9 to 12. A stochastic bit stream is defined in lines 14 to 19. The computation is executed on the ReSC coprocessor in line 22. The results are read in line 25.

6 Experimental Results

We demonstrate the effectiveness of our method on a collection of benchmarks for image processing. We discuss one of these, the gamma correction function, in detail. Then we study the hardware cost and robustness of our architecture on all the test cases.

```
// Original C code for the gamma
// correction function.
...
  x = (uchar)(256*(pow(((float)x)/256,.45)));
...
```

⇓

```
// C code for gamma correction function
// on the the ReSC architecture.

//  Define the Bernstein coefficients:
int ReSC_para[] = {
    1, 99, 738, 357, 1023, 719, 993, 1018
};
...
// Load the coefficients into registers.
for (int i = 0; i < 8; i++) {
    mb_write_datafsl(ReSC_para[i],i);
}
...
// Create a random bitstream.
int rand_data[] = {8, 0};
rand_data[1]=((uint)x)<<2;
for(int i=0; i < 2; i++) {
    mb_write_datafsl(rand_data[i],i);
}
...
// Execute the computation.
while(mb_read_datafsl(0)==0){}
...
// Read the results.
x=(uchar)(mb_read_datafsl(1)>>2);
...
```

Fig. 9: C code fragments for the gamma correction function, $f(x) = x^{0.45}$, on the ReSC architecture.

6.1 A Case Study: Gamma Correction

We continue with our running example, the gamma correction function of Example 2. We present an error analysis and a hardware cost comparison for this function.

6.1.1 Error Analysis

Consider the three error components described in Section 4.

1) **Error due to the Bernstein approximation.** Figure 10 plots the error due to the Bernstein approximation versus the degree of the approximation. The error is measured by (12). It obviously shows that the error decreases as the degree of the Bernstein approximation increases. For a choice of degree $n = 6$, the error is approximately $4 \cdot 10^{-3}$.

   To get more insight into how the error due to the Bernstein approximation changes with increasing degrees, we apply the degree 3, 4, 5 and 6 Bernstein approximations of the gamma correction function to an image. The resulting images for different degrees of Bernstein approximation are shown in Figure 13.

2) **Quantization error.** Figure 11 plots the quantization error versus the number of bits L of the LFSR. In the figure, the x-axis is $1/2^L$, where the range of L is from 5 to 11. For different values of L, b_i' and x' in (14) change according to Equation (9). The quantization error is measured by (14) with the Bernstein polynomial chosen as the degree 6 Bernstein polynomial approximation of the gamma correction function. For each value of L, we evaluate the quantization error on 11 sample points $x = 0, 0.1, \ldots, 0.9, 1$. The mean, the mean plus the standard deviation, and the mean minus the standard deviation of the errors are plotted by a circle, a downward-pointing triangle, and a upward-pointing triangle, respectively.

   Clearly, the means of the quantization error are located near a line, which means that the quantization error is inversely proportional to 2^L. Increasing L will decrease the quantization error.

   To get a better intuitive understanding of how the quantization error changes with increasing number of bits of the LFSR, we apply four different quantizations of the degree-6 Bernstein approximations of the gamma correction function to an image. These four different quantizations are based on LFSRs with 3, 4, 5 and 6 bits, respectively. The resulting images for different degrees of Bernstein approximation are shown in Figure 14.

3) **Error due to stochastic variation.** Figure 12 plots the error due to stochastic variation versus the length N of the stochastic bit stream. In the figure, the x-axis is $1/\sqrt{N}$, where N is chosen to be 2^m, with $m = 7, 8, \ldots, 13$. The error is measured as the average of 50 Monte Carlo simulations of the difference between the stochastic computation result and the quantized implementation of the degree 6 Bernstein polynomial approximation of the gamma correction function. To add the quantization effect, we choose an LFSR of 10 bits. For each N, we evaluate the error on 11 sample points $x = 0, 0.1, \ldots, 0.9, 1$. The mean, the mean plus the standard deviation, and the mean minus the standard deviation of the errors are plotted by a circle, a downward-pointing triangle, and a upward-pointing triangle, respectively.

   The figure clearly shows that the means of the error due to stochastic variation are located near a straight line. Thus, it confirms the fact that the error due to stochastic variation is inversely proportional to $\sqrt{N}$. The error component could be decreased by increasing the length of the stochastic bit stream.

   To have a further impression on how the error due to stochastic variation changes with increasing length of the stochastic bit stream, we apply four stochastic implementations of the gamma correction with varying bit stream lengths to an image. The lengths of the stochastic bit streams in these four implementations are 128, 256, 512 and 1024, respectively. The four stochastic implementations are based on Bernstein approximation of degree 6

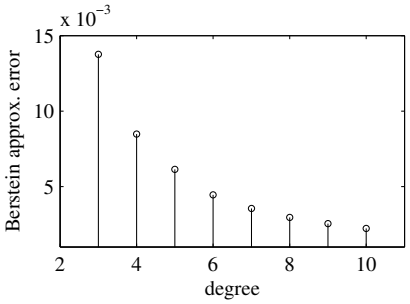


Fig. 10: The Bernstein approximation error versus the degree of the Bernstein approximation.

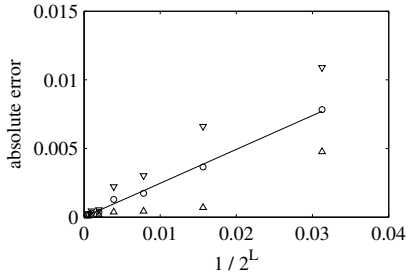


Fig. 11: The quantization error versus $1/2^L$, where L is the number of bits of the pseudo-random number generator. The circles, the downward-pointing triangles, and the upward-pointing triangles represent the means, the means plus the standard deviations, and the means minus the standard deviations of the errors on the sample points $x = 0, 0.1, \ldots, 0.9, 1$, respectively.

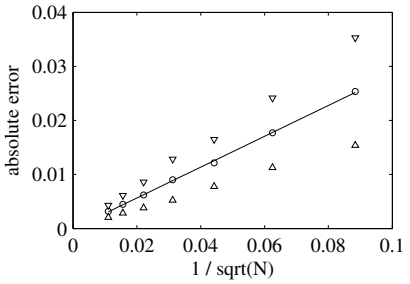


Fig. 12: The error due to random fluctuations versus $1/\sqrt{N}$, where N is the length of the stochastic bit stream. The circles, the downward-pointing triangles, and the upward-pointing triangles represent the means, the means plus the standard deviations, and the means minus the standard deviations of the errors on sample points $x = 0, 0.1, \ldots, 0.9, 1$, respectively.

Fig. 13: The effect of different Bernstein approximation degrees on the gamma correction application. The degrees are (a) 3; (b) 4; (c) 5; (d) 6.

Fig. 14: The effect of different numbers of bits of the LFSR on the gamma correction application. The numbers of bits of the LFSR are (a) 3; (b) 4; (c) 5; (d) 6.

Fig. 15: The effect of different lengths of the stochastic bit streams on the gamma correction application. The lengths of the stochastic bit streams are (a) 128; (b) 256; (c) 512; (d) 1024.

and an LFSR with 10 bits. The resulting images for different lengths of the stochastic bit streams are shown in Figure 15.

6.1.2 Hardware Cost Comparison

The most popular and straight-forward implementation of the gamma correction function is based on direct table lookup. For example, for a display system that supports 8 bits of color depth per pixel, an 8-bit input / 8-bit output table is placed before or after the frame buffer. However, this method is inefficient when more bits per pixel are required. Indeed, for target devices such as medical imaging displays and modern high-end LCDs, 10 to 12 bits per pixel are common. Various methods are used to reduce hardware costs. For example, Lee *et al.* presented a piece-wise linear polynomial (PLP) approximation [20]. They implemented their design on a Xilinx Virtex-4 XC4VLX100-12 FPGA. In order to make a fair comparison, we present implementation results for the same platform.

Table 1 illustrates the hardware cost of the three approaches. The area of the ReSC implementation in-

TABLE 1: Hardware comparisons for three implementations of gamma correction: the direct lookup table method, the piecewise linear polynomial (PLP) approximation method, and our ReSC method.

Input bits	Output bits	Area [slices]		
		Conventional	PLP*	ReSC
8	8	69	–	164
10	10	295	–	177
12	10	486	233	180
13	10	606	242	203
14	10	717	249	236

* Cited from [20]. Extra memory bits are required.

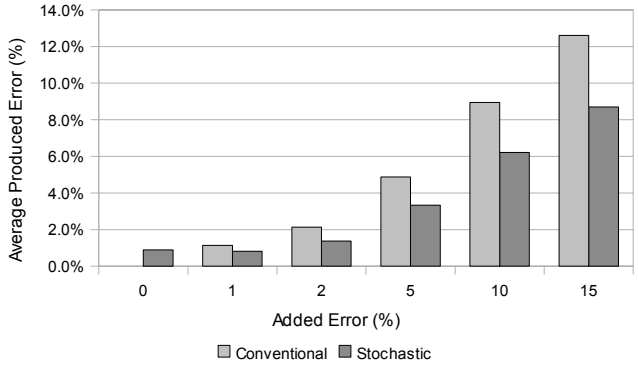


Fig. 16: The result of average output error of conventional and ReSC implementations.

cludes the Randomizer and De-Randomizer units. For the basic 8-bit gamma correction function, our ReSC approach requires 2.4 times the hardware usage of the conventional implementation. For larger number of bits, the hardware usage of our approach increases by only small increments; in contrast, the hardware usage of the conventional implementation increases by a linear amount in the number of bits. In all cases, our approach has better hardware usage than the PLP approximation. Furthermore, our approach provides fault tolerance while the other approaches do not.

6.1.3 Fault Tolerance

To evaluate the robustness of our method, we analyze the effect of soft errors. These are simulated by independently flipping the input bits for a given percentage of the computing elements. For example, if 5% noise was added to the circuit, this implies that 5% of the total number of input bits are randomly chosen and flipped. We compare the effect of soft errors on our implementation to that on conventional implementations.

Figure 16 shows the average percentage error in the output image for five different ratios of added noise. The length of the stochastic stream is fixed at 1024 bits. The stochastic implementation beats the conventional method by less than 2 percent, on average. However, in the conventional implementation, bit flips afflict each bit of the binary radix representation with equal probability. If the most significant bit gets flipped, the error that occurs can be quite large. The analysis of the error distribution is presented in Table 2.

TABLE 2: Analysis of error distribution of the gamma correction function.

Added Error (%)	Produced Error (%)									
	>5		>10		>15		>20		>25	
	Conv.	Stoc.	Conv.	Stoc.	Conv.	Stoc.	Conv.	Stoc.	Conv.	Stoc.
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	4.0	0.0	3.0	0.0	2.4	0.0	2.0	0.0	1.6	0.0
2	8.2	0.5	6.1	0.0	5.0	0.0	4.2	0.0	3.2	0.0
5	19.5	27.8	14.7	1.6	12.1	0.0	10.3	0.0	7.9	0.0
10	35.2	48.4	27.4	24.3	22.7	7.9	19.3	0.0	15.1	0.0
15	48.4	60.1	38.5	36.6	32.1	21.4	27.3	9.1	21.8	0.2

The images in Figure 17 illustrate the fault tolerance of stochastic computation. When soft errors are injected at rate of 15%, the image generated by the conventional method is full of noisy pixels, while the image generated by the stochastic method is still recognizable.

We note that the images become more grey as more error is injected. The reason for this is that, with a stochastic encoding, all errors bring the values closer to center of the unit interval, *i.e.*, a value of $1/2$. For example, consider the situation that the grey level is from 0 to 255 and the length of the stochastic bit stream is 255. Without noise, a purely black pixel, *i.e.*, one with a grey level 0, is ideally represented as a bit stream of all zeroes. If errors are injected, then some of the zeroes in the bit stream become ones; the pixel lightens as its grey level increases. Similarly, without noise, a purely white pixel, *i.e.*, one with a grey level of 255, is ideally represented as a bit stream of all ones. If errors are injected, then some of the ones in the bit stream become zeroes; the pixel darkens as its grey level decreases. For pixels with other grey levels, the trend is similar: injecting errors brings the grey level of pixels toward the mid-brightness value of 128.

6.2 Test Cases

We evaluated our ReSC architecture on ten test cases [23]–[25]. These can be classified into three categories: `Gamma`, `RGB→XYZ`, `XYZ→RGB`, `XYZ→CIE-L*ab`, and `CIE-L*ab→XYZ` are popular color-space converter functions in image processing; `Geometric` and `Rotation` are geometric models for processing two-dimensional figures; and `Example01` to `Example03` are operations used to generate 3D image data sets.

We first compare the hardware cost of conventional deterministic digital implementations to that of stochastic implementations. Next we compare the performance of conventional and stochastic implementations on noisy input data.

6.2.1 Hardware Cost Comparison

To synthesize the ReSC implementation of each function, we first obtained the requisite Bernstein coefficients for it from code written in Matlab. Next, we coded our

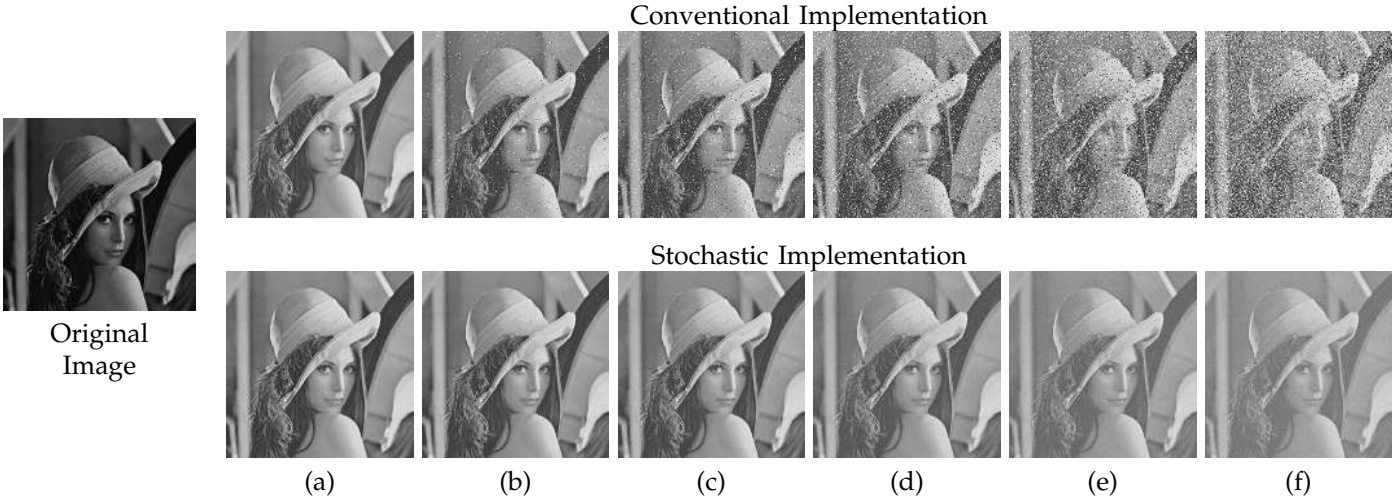


Fig. 17: Fault tolerance for the gamma correction function. The images in the top row are generated by a conventional implementation. The images in the bottom row are generated by our stochastic ReSC implementation. Soft errors are injected at a rate of (a) 0%; (b) 1%; (c) 2%; (d) 5%; (e) 10%; (f) 15%.

TABLE 3: Comparison of the hardware usage (in LUTs) of conventional implementations to our ReSC implementations.

Module	Conventional cost α	ReSC System* cost β	ReSC System* save (%) $(\alpha\text{-}\beta)/\alpha$	ReSC Core** cost γ	ReSC Core** save (%) $(\alpha\text{-}\gamma)/\alpha$
Gamma	96	124	-29.2	16	83.3
RGB→XYZ	524	301	42.6	64	87.8
XYZ→RGB	627	301	52.0	66	89.5
XYZ→CIE-L⋆ab	295	250	15.3	58	80.3
CIE-L⋆ab→XYZ	554	258	53.4	54	90.3
Geometric	831	299	64.0	32	96.1
Rotation	737	257	65.1	30	95.9
Example01	474	378	20.3	46	90.3
Example02	1065	378	64.5	109	89.8
Example03	702	318	54.7	89	87.3
Average	590	286	40.3	56	89.1

* The entire ReSC architecture, including Randomizers and De-Randomizers.
** The ReSC Unit by itself.

reconfigurable ReSC architecture in Verilog, and then synthesized, placed and routed it with Xilinx ISE 9.1.03i on a Virtex-II Pro XC2VP30-7-FF896 FPGA. Table 3 compares the hardware usage of our ReSC implementations to conventional hardware implementations. For the conventional hardware implementations, the complicated functions, e.g., trigonometric functions, are based on the lookup table method. On average, our ReSC implementation achieves a 40% reduction of look-up table (LUT) usage. If the peripheral Randomizer and De-Randomizer circuitry is excluded, then our implementation achieves an 89% reduction of hardware usage.

6.2.2 Fault Tolerance

To study the fault tolerance of our ReSC architecture, we performed experiments injecting soft errors. This consisted of flipping the input bits of a given percentage of the computing elements in the circuit and evaluating the output. We evaluated the output in terms of the average error in pixel values. Table 4 shows the results for three different injected noise ratios for conventional implementations compared to our ReSC implementation of the test cases. The average output error of the conventional implementation is about two times that of the ReSC implementation.

The ReSC approach produces dramatic results when the magnitude of the error is analyzed. In Table 5, we

TABLE 4: The average output error of our ReSC implementation compared to conventional implementations for the color-space converter functions.

Module	Injected Error 1% ReSC	1% Conv.	2% ReSC	2% Conv.	10% ReSC	10% Conv.
Gamma	0.9	0.7	1.6	1.5	7.5	6.8
RGB→XYZ	0.8	2.7	1.4	5.3	6.2	22.4
XYZ→RGB	1.2	3.2	2.3	5.9	8.2	21.6
XYZ→CIE-L⋆ab	0.8	2.1	1.4	3.4	7.3	11.7
CIE-L⋆ab→XYZ	0.8	0.6	1.5	1.2	7.3	7.4
Average	0.9	2.2	1.7	4.0	7.3	15.8

TABLE 5: The percentage of pixels with errors greater than 20% for conventional implementations and our ReSC implementations of the color-space converter functions.

Module	Conventional Injected Error 1%	2%	10%	ReSC Injected Error 1% - 10%
Gamma	1.4	3.8	13.4	0.0
RGB→XYZ	2.2	4.4	20.7	0.0
XYZ→RGB	11.7	20.0	63.8	0.0
XYZ→CIE-L⋆ab	6.1	11.6	43.7	0.0
CIE-L⋆ab→XYZ	2.0	4.0	20.7	0.0
Average	5.0	10.0	37.2	0.0

consider output errors that are larger than 20%. With a 10% soft error injection rate, the conventional approach produces outputs that are more than 20% off over 37% of the time, which is very high. In contrast, our ReSC implementation *never* produces pixel values with errors larger than 20%.

7 Conclusion

In a sense, the approach that we are advocating here is simply a highly redundant, probabilistic encoding of data. And yet, our synthesis methodology is a radical departure from conventional approaches. By transforming computations from the deterministic Boolean domain into arithmetic computations in the probabilistic domain, circuits can be designed with very simple logic. Such stochastic circuits are much more tolerant of errors. Since the accuracy depends only on the statistical distributions of the random bit streams, this fault tolerances scales gracefully to very large numbers of errors.

Indeed, for data intensive applications where small fluctuations can be tolerated, but large errors are catastrophic, the advantage of our approach is dramatic. In our experiments, we never observed errors above 20% with noise injection levels less than 10%, whereas in conventional implementations such errors happened nearly 40% of the time. This fault tolerance is achieved with little or no penalty in cost: synthesis trials show that our stochastic architecture requires less area than conventional hardware implementations.

Because of inherent errors due to random fluctuations, the stochastic approach is best suited for applications that do not require high precision. A serial implementation of stochastic logic, it should be noted, requires relatively many clock cycles to achieve a given precision compared to a conventional implementation: if the resolution of the computation is required to be 2^{-M}, then 2^M clock cycles are needed to obtain the results. However, our stochastic architecture can compute complex functions such as polynomials directly. A conventional hardware implementation typically would implement the computation of such functions over many clock cycles. Accordingly, in an area-delay comparison, the stochastic approach often comes out favorably. Also, a significant advantage of the stochastic architecture is that it can be reconfigured to compute different functions: the function that is computed is determined by the values loaded into the coefficient registers.

In future work, we will develop stochastic implementations for more general classes of functions, such as the multivariate functions needed for complex signal processing operations. Also, we will explore architectures that are tailored to specific domains, such as applications that are data-intensive yet probabilistic in nature and applications that are not probabilistic in nature but can tolerate fluctuations and errors.

References

[1] M. Mitzenmacher and E. Upfal, *Probability and Computing : Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[2] C. H. Papadimitriou, *Computational Complexity*. Addison-Wesley, 1995.

[3] D. T. Gillespie, "A general method for numerically simulating the stochastic time evolution of coupled chemical reactions," *Journal of Computational Physics*, vol. 22, no. 4, pp. 403–434, 1976.

[4] W. Qian and M. D. Riedel, "The synthesis of robust polynomial arithmetic with stochastic logic," in *Design Automation Conference*, 2008, pp. 648–653.

[5] W. Qian, J. Backes, and M. D. Riedel, "The synthesis of stochastic circuits for nanoscale computation," *International Journal of Nanotechnology and Molecular Computation*, vol. 1, no. 4, pp. 39–57, 2010.

[6] B. Gaines, "Stochastic computing systems," in *Advances in Information Systems Science*. Plenum, 1969, vol. 2, ch. 2, pp. 37–172.

[7] S. Toral, J. Quero, and L. Franquelo, "Stochastic pulse coded arithmetic," in *International Symposium on Circuits and Systems*, vol. 1, 2000, pp. 599–602.

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

[9] J. von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," in *Automata Studies*. Princeton University Press, 1956, pp. 43–98.

[10] E. F. Moore and C. E. Shannon, "Reliable circuits using less reliable relays," *Journal of the Franklin Institute*, vol. 262, pp. 191–208, 281–297, 1956.

[11] H. Chang and S. Sapatnekar, "Statistical timing analysis under spatial correlations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 9, pp. 1467–1482, 2005.

[12] D. Beece, J. Xiong, C. Visweswariah, V. Zolotov, and Y. Liu, "Transistor sizing of custom high-performance digital circuits with parametric yield considerations," in *Design Automation Conference*, 2010, pp. 781–786.

[13] K. Nepal, R. Bahar, J. Mundy, W. Patterson, and A. Zaslavsky, "Designing logic circuits for probabilistic computation in the presence of noise," in *Design Automation Conference*, 2005, pp. 485–490.

[14] K. Palem, "Energy aware computing through probabilistic switching: A study of limits," *IEEE Transactions on Computers*, vol. 54, no. 9, pp. 1123–1137, 2005.

[15] S. Narayanan, J. Sartori, R. Kumar, and D. Jones, "Scalable stochastic processors," in *Design, Automation and Test in Europe*, 2010, pp. 335–338.

[16] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. Hoe, "Multi-bit error tolerant caches using two-dimensional error coding," in *International Symposium on Microarchitecture*, 2007, pp. 197–209.

[17] X. Li, W. Qian, M. D. Riedel, K. Bazargan, and D. J. Lilja, "A reconfigurable stochastic architecture for highly reliable computing," in *Great Lakes Symposium on VLSI*, 2009, pp. 315–320.

[18] W. Qian, M. D. Riedel, K. Barzagan, and D. Lilja, "The synthesis of combinational logic to generate probabilities," in *International Conference on Computer-Aided Design*, 2009, pp. 367–374.

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

[20] D. Lee, R. Cheung, and J. Villasenor, "A flexible architecture for precise gamma correction," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 4, pp. 474–478, 2007.

[21] J. Ortega, C. Janer, J. Quero, L. Franquelo, J. Pinilla, and J. Serrano, "Analog to digital and digital to analog conversion based on stochastic logic," in *International Conference on Industrial Electronics, Control, and Instrumentation*, 1995, pp. 995–999.

[22] H. P. Rosinger, *Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link Channel*, Xilinx Inc., 2004. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp529.pdf

[23] Irotek, "EasyRGB," 2008. [Online]. Available: http://www.easyrgb.com/index.php?X=MATH

[24] D. Phillips, *Image Processing in C*. R & D Publications, 1994.

[25] T. Urabe, "3D examples," 2002. [Online]. Available: http://mathmuse.sci.ibaraki.ac.jp/geom/param1E.html