A Reconfigurable Stochastic Architecture for Highly Reliable Computing*

Xin Li, Weikang Qian, Marc D. Riedel, Kia Bazargan, and David J. Lilja
Department of Electrical and Computer Engineering, University of Minnesota
Minneapolis, Minnesota, USA
{lixxx914, qianx030, mriedel, kia, lilja}@umn.edu

ABSTRACT

Mounting concerns over variability, defects and noise motivate a new approach for integrated circuits: the design of stochastic logic, that is to say, digital circuitry that operates on probabilistic signals, and so can cope with errors and uncertainty. Techniques for probabilistic *analysis* are well established. We advocate a strategy for *synthesis*. In this paper, we present a reconfigurable architecture that implements the computation of arbitrary continuous functions with stochastic logic. We analyze the sources of error: approximation, quantization, and random fluctuations. We demonstrate the effectiveness of our method on a collection of benchmarks for image processing. Synthesis trials show that our stochastic architecture requires less area than conventional hardware implementations. It achieves a large speed up compared to software conventional implementations. Most importantly, it is much more tolerant of soft errors (bit flips) than these deterministic implementations.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Data-flow architectures*; B.8.1 [**Performance and Reliability**]: Reliability, Testing and Fault-Tolerance

General Terms

Design, Performance, Reliability

Keywords

Stochastic logic, Reconfigurable architecture, Reliable computing

1. INTRODUCTION

The successful design methodology 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 a circuit is fixed; 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 techniques at the physical level. As the scale of technology pushes into ever deeper regimes, the cost and complexity are threatening to stall progress. We argue that the deterministic paradigm is untenable.

*This work is supported by a grant from the MICRO Focus Center Research Program on Functional Engineered Nano-Architectonics and a grant from Intel Corporation.

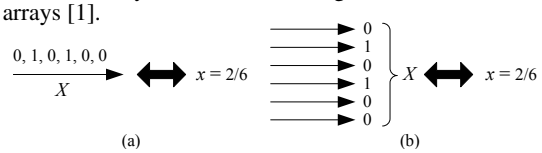
GLSVLSI'09, May 10–12, 2009, Boston, Massachusetts, USA.

1.1 Stochastic Logic

We advocate a novel view for computation: instead of synthesizing circuits that transform definite inputs into definite outputs – say Boolean, integer, or floating-point values into the same – we synthesize circuits that transform probability values into probability values. 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 [1].



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: $P(X = 1) = x$.

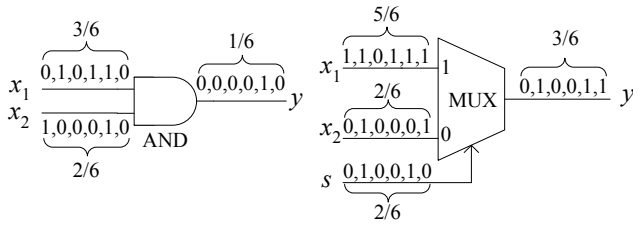
Our synthesis strategy is to cast logic computations as arithmetic operations in the probabilistic domain and implement these directly as stochastic operations on data-paths. Two simple arithmetic operations – multiplication and scaled addition – are illustrated in Figure 2.

- **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
$$y = P(Y = 1) = P(X_1 = 1 \text{ and } X_2 = 1)$$
$$= P(X_1 = 1)P(X_2 = 1) = x_1 x_2.$$
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.



(a) (b)

2: Implementation of multiplication and scaled addition: (a) Multiplication; (b) Scaled addition.

With stochastic logic, if the resolution of a computation is required to be 2^{-M}, then the length of the bit stream should be 2^M. While the method entails redundancy in the encoding of signal values, complex operations can be performed using simple logic. The advantage of a stochastic architecture 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 cause 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 minuscule 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:

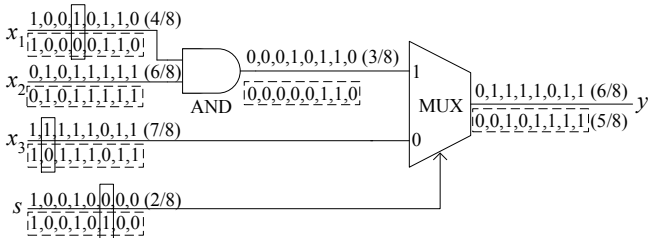
$$y = x_1x_2s + x_3(1-s).$$

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 $5/8$.

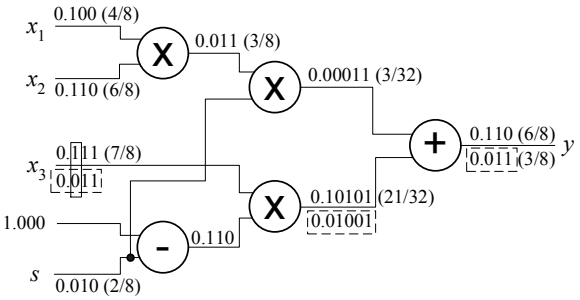
In contrast, Figure 3(b) shows a conventional implementation of the function with multiplication and addition modules operating on a binary radix representation: the real numbers $x_1 = 4/8$, $x_2 = 6/8$, $x_3 = 7/8$, and $s = 2/8$ are encoded as $(0.100)_2$, $(0.110)_2$, $(0.111)_2$, and $(0.010)_2$, 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.

1.2 Related Work and Context

The topic of computing reliably with unreliable components dates back to von Neumann and Shannon [2, 3]. 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.



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



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

3: An illustration of the fault tolerance of stochastic logic.

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. Many flavors of probabilistic design have been proposed for integrated circuits. For instance, [4] presents a design methodology based on Markov random fields geared toward nanotechnology; [5] presents a methodology based on probabilistic CMOS, with a focus on energy efficiency.

On the one hand, our approach is more narrowly circumscribed: we focus on synthesizing circuits at the logic level, using ordinary elements (AND, OR, MUX, etc.) without explicit reference to the underlying technology. On the other hand, our aim is broader: we present a complete synthesis methodology for circuitry that computes in terms of statistical distributions. The resulting logic 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 precision of the statistics.

Fuzzy logic is similar, conceptually, to our notion of stochastic logic. However, in fuzzy logic, a real value in the unit interval represents the degree of truth, which is different from a probability [6]. Thus, fuzzy logic is tool for reasoning; in contrast, we use stochastic logic in the context of circuit computation.

Early work on the topic of stochastic logic established how arithmetic operations like multiplication, addition, and division can be implemented [7, 8]. In prior work, we presented a technique for synthesizing stochastic logic for the computation of polynomials [9]. The method is based on converting polynomials into a particular mathematical form – Bernstein polynomials – and then implementing the computation with generalized multiplexing. In this paper, we present a reconfigurable architecture that implements the computation of arbitrary continuous functions with stochastic logic.

2. THE RECONFIGURABLE ARCHITECTURE

2.1 Synthesizing Arbitrary Continuous Functions

2.1.1 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 smaller than zero or larger than one, e.g., $1.2x - 1.2x^2$, since we cannot represent such coefficients with stochastic bit streams.

In [9], we proposed a method for implementing arbitrary polynomials, including those with coefficients 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, we can synthesize stochastic logic that implements it. The procedure begins by transforming a power-form polynomial into a Bernstein polynomial [10]. 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 not 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. Such a polynomial can be implemented stochastically based on multiplexing [9].

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 coefficients of this Bernstein polynomial are in the unit interval.

2.1.2 Synthesizing Non-Polynomial Functions

It was proved in [9] that stochastic logic can *only* implement polynomial functions. In real applications, of course, we often encounter non-polynomial functions, such as trigonometric functions. In this section, we present a method for synthesizing stochastic logic that implements arbitrary functions approximately via Bernstein polynomials. We formulate the problem as follows:
Given $f(x)$, a continuous function on the unit interval, and n, the degree of a Bernstein polynomial, find real numbers b_i, $i = 0, \ldots, n$, that minimize

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

subject to

$$0 \le b_i \le 1, \text{ for all } i = 0, 1, \ldots, n. \tag{4}$$

Here we try to find the optimal Bernstein polynomial approximation by minimizing an objective function, Equation (3), that measures the approximation error. This is the square of the L^2 norm on the difference between the original function $f(x)$ and the Bernstein polynomial $B(x) = \sum_{i=0}^{n} b_i B_{i,n}(x)$. The integral is on the unit interval because x, representing a probability value, is always in the unit interval. The constraints in Equation (4), guarantee that the Bernstein coefficients are all in the unit interval, so that the function can be implemented by stochastic logic.

If we expand (3), then an equivalent objective function is

$$f(\boldsymbol{b}) = \frac{1}{2}\boldsymbol{b}^T \boldsymbol{H} \boldsymbol{b} + \boldsymbol{c}^T \boldsymbol{b}, \tag{5}$$

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.

Example 2
Consider the non-polynomial function

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

(This is the gamma correction function; it is discussed in detail in Section 3.1.) We approximate this function by a Bernstein polynomial of degree 6. By solving the constrained quadratic optimization problem, we obtain the Bernstein coefficients:

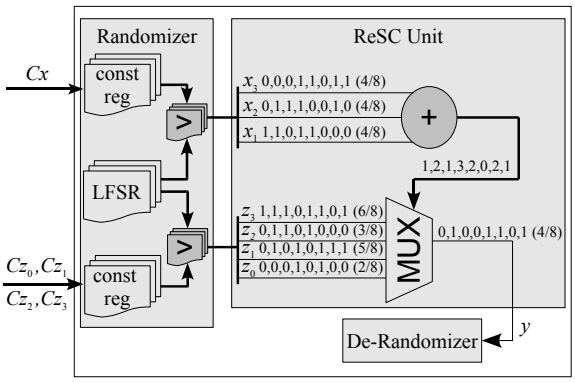
$$b_0 = 0.0955, b_1 = 0.7207, b_2 = 0.3476, b_3 = 0.9988,$$
$$b_4 = 0.7017, b_5 = 0.9695, b_6 = 0.9939$$

2.2 The ReSC Architecture

As illustrated in Figure 4, our reconfigurable stochastic architecture (ReSC) is 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.

2.2.1 The ReSC Unit

The ReSC Unit is the kernel of the architecture. It implements Bernstein polynomials with coefficients in the unit interval. As described in Section 2.1, we use this to implement arbitrary continuous functions.



4: 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 $x = 0.5$

A Bernstein polynomial is implemented by stochastic logic via "generalized multiplexing" [9]. As illustrated in Figure 4, this consists of an adder block and a multiplexer block. The adder block counts the number of ones in an input set $\{x_1, \ldots, x_n\}$. The selecting inputs to the multiplexing block are the outputs of the adder block; the data inputs to the multiplexer are $z_0, \ldots, z_n$, corresponding to the Bernstein coefficients. The output of the multiplexer y is set to be z_i $(0 \le i \le n)$, where i equals the binary number computed by the adder.

The stochastic input bit streams to the ReSC Unit are configured as follows:

- The inputs $x_1, \ldots, x_n$ are *independent* stochastic bit streams $X_1, \ldots, X_n$ that represents the probabilities $P(X_i = 1) = x \in [0, 1]$, for $1 \le i \le n$. The probability x in these bit streams is controlled by the binary number C_x in the constant register, as shown in Figure 4. The constant register is a part of the Randomizer Unit, discussed below.

- 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 $0 \le i \le n$, where the b_i's are Bernstein coefficients. 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 the constant registers, as shown in Figure 4.

The output of the ReSC Unit is a stochastic bit stream Y corresponding to y, the optimal Bernstein approximation of the target function.

Figure 4 also shows an instance of an ReSC Unit that implements the function $f_1(x)$ from Example 1, evaluated at $x = 0.5$. The stochastic bit streams X_1, X_2 and X_3 are independent and each represent the probability value $x = 0.5$. The stochastic bit streams $Z_0, \ldots, Z_3$ represent probabilities $P(Z_0 = 1) = \frac{2}{8}$, $P(Z_1 = 1) = \frac{5}{8}$, $P(Z_2 = 1) = \frac{3}{8}$, and $P(Z_3 = 1) = \frac{6}{8}$. As expected, the ReSC Unit computes the correct output value: $f_1(0.5) = 0.5$.

2.2.2 The Randomizer and the De-Randomizer

The Randomizer Unit, shown in Figure 4, is used to generate the stochastic input bit streams. It consist of a pseudo-random number generator, such as a linear feedback shift register (LFSR), that generates a new number R on each clock cycle. This number is compared to a constant input value C; if smaller, it generates a one as the next bit in the stream; otherwise, it generates a zero. If we use an LFSR with L bits, we obtain pseudo-random numbers from the set $\{1, 2, \ldots, 2^L - 1\}$. Accordingly, the probabilities values corresponding to the bit stream are *discrete* values from the set $S = \{0, \frac{1}{2^L-1}, \ldots, 1\}$; hence there is quantization error. We map a given number p to the closest number in S. Thus, C is determined as

$$C = \text{round}(p(2^L - 1)) + 1, \tag{6}$$

where round(x) is the integer closest to x.

The De-Randomizer Unit translates the result of the computation, expressed as a stochastic bit stream, into numerical form. This is implemented with a counter that tallies the number of ones that it sees during the observation interval, and normalizes the count by the length of the interval to get the fractional output.

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 [11]. Furthermore, the total area of the Randomizer and De-Randomizer units could be amortized in some applications (*e. g.*, DSP), which have multiple computations in series.

2.3 Error Analysis

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

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

$$e_1 = \left| f(x) - \sum_{i=0}^{n} b_i B_{i,n}(x) \right|. \tag{7}$$

We can use the average L^2-norm to measure this:

$$e_{1\text{avg}} = \left(\int_0^1 (f(x) - \sum_{i=0}^{n} b_i B_{i,n}(x))^2 \, \mathrm{d}x \right)^{0.5} \tag{8}$$

This error only depends on the original function $f(x)$ and on the degree of the Bernstein polynomial; it decreases with increasing n.

2. **Quantization error**: This is the round-off error that occurs when mapping real numbers, corresponding to probabilities, to pseudo-random bit streams. Assume that the pseudo-random number generator has L bits. Since we map a given value p to the closest value p' in the discrete set $S = \{0, \frac{1}{2^L-1}, \ldots, 1\}$, we will compute $\sum_{i=0}^{n} b'_i B_{i,n}(x')$ instead of $\sum_{i=0}^{n} b_i B_{i,n}(x)$, where b'_i and x' are the closest numbers to b_i and x in S, respectively. Thus, the quantization error is

$$e_2 = \left| \sum_{i=0}^{n} b'_i B_{i,n}(x') - \sum_{i=0}^{n} b_i B_{i,n}(x) \right|. \tag{9}$$

Mathematically, we can deduce an upper bound on this error:

$$e_2 \leq \frac{n+1}{2(2^L - 1)}, \tag{10}$$

where n is the degree of the Bernstein approximation. Thus, the quantization error is inversely proportional to 2^L. We can mitigate it by increasing the number of bits L of the LFSR.

3. **Error due to random fluctuations**: Due to the Bernstein approximation and the quantization effect, the bits in the output stream $Y(t)$ $(t = 1, 2, \ldots, N)$ have probability $q' = \sum_{i=0}^{n} b'_i B_{i,n}(x')$ of being one. The De-Randomizer measures the rate of ones in the output stochastic bit stream:

$$y = \frac{1}{N} \sum_{t=1}^{N} Y(t). \tag{11}$$

It is easily seen that the expectation of y is $E[y] = q'$. However, because of random fluctuations, the rate of ones in the output stream y might not be exactly equal to q'. This error can be measured by the variance:

$$\begin{aligned} E[(y - q')^2] &= E[(y - E[y])^2] = Var[y] \\ &= Var[\frac{1}{N} \sum_{t=1}^{N} Y(t)] = \frac{q'(1 - q')}{N}. \end{aligned} \tag{12}$$

Thus, the error due to random fluctuations is

$$e_3 = |y - q'| \approx \sqrt{\frac{q'(1 - q')}{N}}. \tag{13}$$

It is inversely proportional to $\sqrt{N}$, and so decreases with increasing length of the bit stream.

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

$$e = |f(x) - y| \leq e_1 + e_2 + e_3. \tag{14}$$

3. EXPERIMENTAL RESULTS

We demonstrate the effectiveness of our method on a collection of benchmarks for image processing. We describe the design of one of these, the gamma correction function, in detail. Then we analyze the performance and robustness of our architecture on all the test cases.

3.1 A Case Study: 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:

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

where V_{in} is normalized over zero and one [12]. We apply a value of $\gamma = 0.45$, which is the value used in most TV cameras.

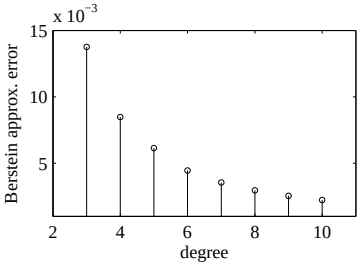
3.1.1 Synthesis

The coefficients of the Bernstein approximation of degree 6 for the gamma correction function were given in Example 2. In our implementation, the LFSR has 10 bits. Thus, by (6), the numbers that we load into in the constant coefficient registers are:

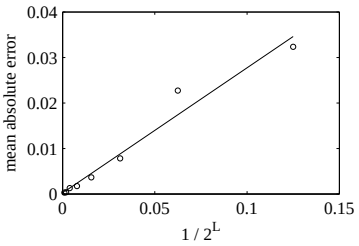
$$C_0 = 99, C_1 = 738, C_2 = 357, C_3 = 1023,$$
$$C_4 = 719, C_5 = 993, C_6 = 1018.$$

3.1.2 Error Analysis

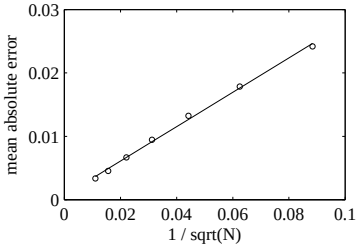
The error due to the Bernstein approximation, measured by (8), versus the degree of approximation is shown in Figure 5. The quantization error, measured by (9) and averaged on 11 evaluation points $x = 0, 0.1, \ldots, 0.9, 1$, versus the number of bits L of the LFSR is shown in Figure 6. The values on the x-axis are $1/2^L$, for values of



5: The Bernstein approximation error versus the degree of the Bernstein approximation.



6: The quantization error versus $1/2^L$, where L is the number of bits of the pseudo-random number generator.



7: The error due to random fluctuations versus $1/\sqrt{N}$, where N is the length of the stochastic bit stream.

L from 3 to 10. Clearly, the quantization error is inversely proportional to 2^L. The error due to random fluctuations, averaged on 11 evaluation points $x = 0, 0.1, \ldots, 0.9, 1$, versus the length N of the stochastic bit stream is shown in Figure 7. The values on the x-axis are $1/\sqrt{N}$, where N is chosen to be 2^m, with $m = 7, 8, \ldots, 13$. The figure clearly shows that the error due to random fluctuations is inversely proportional to $\sqrt{N}$.

3.1.3 Hardware Cost Comparison

The most popular and straight-forward implementation of the gamma correction function is based on direct table lookups. 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 [12]. 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.

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 [12]. Extra memory bits are required.

Table 1 illustrates the hardware cost of the three approaches. The area of ReSC also includes 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 numbers 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 approach.

3.2 Test Cases

We evaluated our ReSC architecture on ten test cases [13, 14, 15]. 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 obtained the requisite coefficients for our ReSC implementation of these functions from code written in Matlab.

3.2.1 Hardware Cost Comparison

We coded our 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 2 compares the hardware usage of our ReSC implementations to conventional hardware implementations. On average, our ReSC implementation achieves a 40% reduction of LUT usage. If the peripheral Randomizer and De-Randomizer circuitry is excluded, then our implementation achieves an 89% reduction of hardware usage.

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

Module	Conventional cost α	ReSC			
		System*		Core**	
		cost β	save (%) $(\alpha$-$\beta)/\alpha$	cost γ	save (%) $(\alpha$-$\gamma)/\alpha$
`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.

3.2.2 Software Performance Comparison

For a software implementation, we chose the MicroBlaze 32-bit soft COTS RISC processor core, version 6.00b, with an FPU and with a 16KB cache. This processor core was mapped onto the same FPGA platform as above. It occupied 5386 LUTs. We chose 1024 cycles as the default execution time for our stochastic computation. We compared this to conventional implementations of the test functions, specified as C programs. These were compiled for the MicroBlaze core via gcc version 3.4.1, using the default level 2 optimization.

3: The execution time (in clock cycles) for software implementations and the speedup of our ReSC implementation.

Module	*Math*		*Optimized*	
	time (x10^3)	ReSC speedup	time (x10^3)	ReSC speedup
`Gamma`	2354	2223	0.15	0.14
`RGB→XYZ`	7747	7315	793	748
`XYZ→RGB`	7601	7177	912	861
`XYZ→CIE-L*ab`	7147	6749	372	351
`CIE-L*ab→XYZ`	14043	13261	530	500
`Geometric`	1860	1757	145	137
`Rotation`	1927	1820	101	95
`Example01`	3503	3308	61	57
`Example02`	5211	4920	102	96
`Example03`	2398	2264	113	107
Average	5379	5079	313	312

We use 100MHz, the maximum clock frequency of the MicroBlaze processor, as the frequency for both implementations. The improvement in the execution time with our ReSC approach is shown in Table 3. Two different methods of programming the test functions were tried. The first method, labeled "*Math*" in the table, uses the standard functions from the "`math.h`" library. This required just a few lines of C codes. However, with this library, the operations are performed in floating-point mode, which degrades

the overall performance. Hence, we also optimized the functions by using table lookups. This approach is labeled "*Optimized*" in the table. Compared with the conventional software implementations, our ReSc approach speeds up the computation by factors of hundreds or thousands.

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

Module	Injected Error					
	1%		2%		10%	
	ReSC	Conv.	ReSC	Conv.	ReSC	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

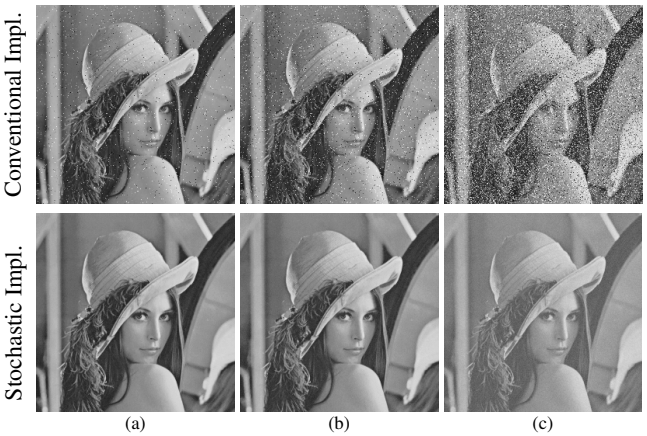
5: The percentage of pixels with errors greater than 20% for conventional implementations and our ReSC implementations of the color-space converter functions.

Module	Conventional Injected Error			ReSC Injected Error
	1%	2%	10%	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

3.2.3 Fault Tolerance

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. To study the fault tolerance of our ReSC architecture, we performed experiments injecting soft errors. This consisted of flipping the input and output bits of a given percentage of the computing elements in the circuit and evaluating the output. For example, if 2% noise was injected, this implies that 2% of the total number of bits on wires in the circuit were chosen randomly and flipped. 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 errors is analyzed. In Table 5, we 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% the time. The result: nearly unreadable images. In contrast, our ReSC implementation *never* produces pixel values with errors larger than 20%. Figure 8 shows, visually, what a difference this makes.



8: Results of error injection for implementations of the gamma correction function. Soft errors are injected at a rate of: (a) 1%; (b) 2%; (c) 10%.

4. CONCLUSION AND FUTURE WORK

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, we achieve circuit designs that 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. We showed that we can implement computation that *never* produces errors greater than 20%, compared to a conventional implementation that produces such errors 37% of the time. This fault tolerance is achieved with little or no penalty in cost or performance: synthesis trials show that our stochastic architecture requires less area than conventional hardware implementations. It achieves a large speed-up compared to software implementations.

One might argue that the conventional hardware accelerators could get further speedup, or even might achieve better cost-performance characteristic if evaluated by delay-area metric. However, compared with the fixed conventional hardware accelerators, the ReSC architecture could be reconfigured for multi computation functions. Furthermore, for the potential applications which our approach targets, *e. g.*, the real-time embedded applications, the system just need to guarantee that deadlines can be met, and faster speed does not imply better results. Hence, our ReSC approach would yield a better trade-off between cost and performance.

In future work, we will study the trade-offs between computation time, accuracy, and reliability for stochastic implementations of larger designs, including a complete Open RISC processor. Also, we will explore architectures that are tailored to specific domains, such as scientific computing, targeting applications that are data-intensive yet statistical in nature.

5. REFERENCES

[1] W. Qian, J. Backes, and M. Riedel, "The synthesis of stochastic circuits for nanoscale computation," in *IWLS '07*, pp. 176–183.

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

[3] E. F. Moore and C. E. Williams, "Reliable circuits using less reliable relays," *J. Franklin Inst.*, vol. 262, pp. 191–208, 281–297, 1956.

[4] K. Nepal, R. Bahar, J. Mundy, W. Patterson, and A. Zaslavsky, "Designing logic circuits for probabilistic computation in the presence of noise," in *DAC '05*, pp. 485–490.

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

[6] C. C. Lee, "Fuzzy logic in control systems: Fuzzy logic controller – part I," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 20, no. 2, pp. 404–418, 1990.

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

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

[9] W. Qian and M. Riedel, "The synthesis of robust polynomial arithmetic with stochastic logic," in *DAC '08*, pp. 648–653.

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

[11] 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 *IECON'95*, pp. 995–999.

[12] D.-U. Lee, R. C. C. Cheung, and J. D. Villasenor, "A flexible architecture for precise gamma correction," *IEEE Trans. VLSI Syst.*, vol. 15, no. 4, pp. 474–478, 2007.

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

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

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