Effect of LFSR Seeding, Scrambling and Feedback Polynomial on Stochastic Computing Accuracy

Jason H. Anderson[1], Yuko Hara-Azumi[2], Shigeru Yamashita[3]
[1]Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, Ontario, Canada
[2]Dept. of Communications and Computer Engineering, Tokyo Institute of Technology, Tokyo, Japan
[3]Dept. of Computer Science, Ritsumeikan University, Shiga, Japan
Email: janders@ece.toronto.edu, hara@cad.ce.titech.ac.jp

***Abstract*—Stochastic computing (SC) [1] has received attention recently as a paradigm to improve energy efficiency and fault tolerance. SC uses hardware-generated random bitstreams to represent numbers in the [0:1] range – the number represented is the probability of a bit in the stream being logic-1. The generation of random bitstreams is typically done using linear-feedback shift register (LFSR)-based random number generators. In this paper, we consider how best to design such LFSR-based stochastic bitstream generators, as a means of improving the accuracy of stochastic computing. Three design criteria are evaluated: 1) LFSR seed selection, 2) the utility of scrambling LFSR output bits, and 3) the LFSR polynomials (i.e. locations of the feedback taps) and whether they should be unique vs. uniform across stream generators. For a recently proposed multiplexer-based stochastic logic architecture [8], we demonstrate that careful seed selection can improve accuracy results vs. the use of arbitrarily selected seeds. For example, we show that stochastic logic with seed-optimized 255-bit stream lengths achieves accuracy better than that of using 1023-bit stream lengths with arbitrary seeds: an improvement of over 4× in energy for equivalent accuracy.**

I. Introduction

Stochastic computing (SC) is a style of approximate computing that uses randomly generated bitstreams to represent numbers. Numerical values in stochastic computing are locked to the [0:1] range, where a value x is represented by a random bitstream having the property that the probability of each bit being logic-1 is x. Fault tolerance is improved relative to a traditional binary number representation because each bit in a stochastic bitstream has equal weight – bitflips due to faults have little impact on the overall value of the bitstream. Moreover, certain types of computations are lightweight to implement using a stochastic representation in comparison with traditional binary, potentially offering area and power benefits (discussed below). Recent applications of SC include image processing [6] and numerical integration [10]. Stochastic logic circuits use random number generators to produce bitstreams with specific values, which are then fed into circuits that perform the desired computations in the stochastic representation. As the input bitstreams are *random*, the results of stochastic computing are not guaranteed to be exact. In this paper, we explore methodologies to raise the accuracy of stochastic computing through design decisions pertaining to the bitstream generators.

Fig. 1(a) shows an AND gate receiving two stochastic streams with values a and b as input and producing a stream z at its output. Assuming streams a and b are statistically independent, the function computed by the circuit is: $z = a \cdot b$ (multiplication). Fig. 1(b) shows a 2-to-1 multiplexer, which can be used to implement scaled addition: $z = s \cdot a + (1-s) \cdot b$. Fig. 1(c) shows the circuit typically used to generate a stochastic bitstream. In this case, a 255-bit-length stream[1] is

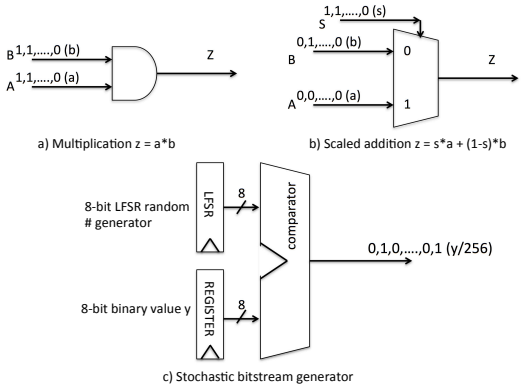

Fig. 1. Stochastic logic structures.

generated by an 8-bit linear-feedback shift register (LFSR) random number generator, whose output value is compared on each cycle with an 8-bit binary value y. On each cycle, the comparator produces 1 if the LFSR value is less than y, yielding a stochastic bitstream with the value $y/255$.

The accuracy of stochastic computing generally hinges on the random bitstreams being statistically independent from one another. However, the LFSRs used in stochastic logic circuits are pseudo-random number generators and cycle through numbers in a fixed repeating sequence. The usual method to approximate independence is to seed (initialize) LFSRs to start at different states. In such an approach, while the various LFSRs are stepping through the same numbers in the same order, they are all at different points in the overall sequence. One aspect of LFSRs explored in this paper is whether there exists a "good" set of starting points for the LFSRs, as opposed to selecting them arbitrarily.

Even when different starting points are used, the values produced by stochastic bitstream generators may exhibit correlations detrimental to the independence goal, possibly owing to the underlying LFSRs traversing states in the same order. To combat this, we consider two easy-to-implement LFSR design alternatives. The first approach is to permute the output bits of each LFSR in the system; i.e. each LFSR is given a different (yet fixed) output bit permutation, thereby making the state order appear different. The second approach we consider is the use of a diverse set of feedback polynomials (tap locations) across LFSR instances, rather than the use of a single polynomial for all LFSRs. Both of these approaches break the fixed-state-ordering property and both are relatively "cheap" to implement in hardware.

We evaluate the seed selection and design alternatives empirically, using a recently proposed multiplexer-based SC architecture [8]. The contributions of this paper are as follows:

[1]The stream length is 255 instead of 256, as the 0-state in the LFSR is normally not used.

- Evaluation on SC accuracy of: 1) permuting LFSR output bits; 2) the use of diverse LFSR feedback polynomials.
- Monte Carlo-style seed sweeping for SC: 1) an approach that is *independent* of the function being implemented in the stochastic domain, and 2) an approach that is tied to the specific function.
- A study demonstrating that SC accuracy is improved by both the design alternatives and seed selection.
- An analysis of energy and performance of an FPGA-based stochastic logic implementation.

While the LFSR design alternatives considered here are certainly not new, the impact of LFSR design and seeding on SC accuracy has not appeared in prior literature, nor has prior work analyzed SC energy in the FPGA context. While it is perhaps unsurprising that seeding has an effect on accuracy, the extent of its impact is shown to be considerable such that with seeding, shorter streams (less precision) can be used in lieu of longer streams to achieve the same or better accuracy. For a given accuracy level, seed sweeping provides 4× energy reduction (owing to the ability to use shorter streams) vs. the use of arbitrary seeds.

II. Background

A key concept in SC is the notion of *precision*. Consider a bitstream of length l, where $l = 2^Q - 1$, Q being a natural number. The number of bits in the stream that may be logic-1 ranges from 0 to *at most* $2^Q - 1$: its range is from [0:$2^Q - 1$]. The stochastic bitstream is therefore said to have Q-bit precision. Increasing from Q to $Q + 1$-bit precision requires doubling the stream length. In SC, linear increases in precision require exponentially more time.

LFSRs are typically used as random number generators in SC implementations, as they are inexpensive, consuming one flip-flop per bit, as well as few logic gates. The idea to judiciously seed LFSRs to produce better results has also been applied in other domains, notably in built-in-self-test to improve fault coverage using fewer vectors, e.g. [13], [3].

A. Multiplexer-Based Stochastic Logic Architecture

The focus in this paper is on the SC structure illustrated in Fig. 2 – a MUX-based stochastic logic circuit proposed by Qian et al. [8] that can approximate *any* continuous function with a [0:1] domain and [0:1] range. Observe that S drives the select inputs of the multiplexer. S has $\lceil log_2 n \rceil$ bits, and on each clock cycle, S is the sum of individual bits from n stochastic streams, each representing the value x. Thus, S may take on any value from 0 (all bits 0) up to n (all bits 1). If the streams are independent, the probability of S being a particular value i is given by the binomial distribution:

$$P(S = i|x) = \sum_{i=0}^{n} \binom{n}{i} \cdot x^i \cdot (1-x)^{n-i} \quad (1)$$

The data inputs of the MUX are driven by streams with different values, $b_0 \ldots b_n$. Each clock cycle, a bit from one of the b streams is selected (based on S) to be passed to output z. The probability of a particular bit, b_i being selected is $P(S = i|x)$, as given by (1). Hence, the overall function computed by the structure is:

$$z(x) = \sum_{i=0}^{n} b_i \cdot \binom{n}{i} \cdot x^i \cdot (1-x)^{n-i} \quad (2)$$

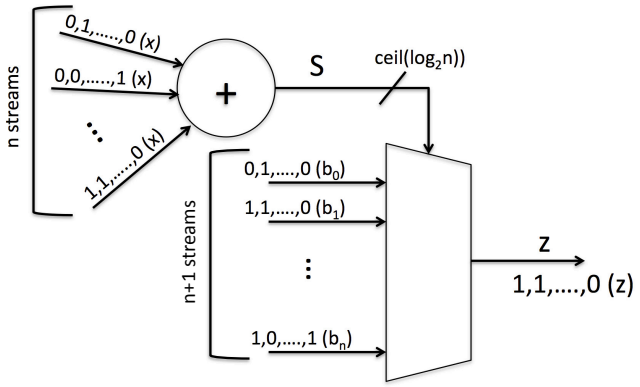

Fig. 2. MUX-based stochastic computing architecture [8].

which is called a Bernstein polynomial [7] of order n, with the b_i's being the Bernstein coefficients. Bernstein polynomials can approximate *any* continuous function in the [0:1] range, with the approximation becoming exact as $n \to \infty$. The Bernstein coefficients for a desired function, $f(x)$, can be computed as described in [8] using MATLAB by finding the b_i values that minimize: $\phi = \int_0^1 (f(x) - \sum_{i=0}^{n} b_i \cdot \binom{n}{i} \cdot x^i \cdot (1-x)^{n-i})^2 dx$, i.e. the sum of the squared error between $f(x)$ and its Bernstein approximation.

For a given target function, the streams $b_0 \ldots b_n$ represent constant coefficients and prior work has proposed using combinational [9] and sequential [11] circuits to generate such "constant" stochastic streams using a limited number of LFSRs. We expect the techniques proposed here are also compatible with such work. However, one of the advantages of the MUX-based SC architecture is the target function can be changed flexibly during system operation by making the $b_0 \ldots b_n$ values *programmable* via an instance of the circuit in Fig 1(c) for generating each b_i – this is the scenario assumed in this paper.

III. Seed Selection for LFSRs

To motivate the impact of LFSR seeding on SC results, consider two stochastic streams in 4-bit precision: stream a representing 3/16 and stream b representing 7/16. To produce stream a requires an LFSR whose output is checked to be less than 4. Example LFSR output in hex and the corresponding stream a is as follows:

```
LFSR: F,7,3,1,8,4,2,9,C,6,B,5,A,D,E
a:    0,0,1,1,0,0,1,0,0,0,0,0,0,0,0
```

Now consider the same LFSR, but seeded differently to produce stream b (the LFSR output is compared with 8). In this case, the LFSR is seeded with 4, instead of F. The LFSR steps through the same sequence, but the start point is different.

```
LFSR: 4,2,9,C,6,B,5,A,D,E,F,7,3,1,8
b:    1,1,0,0,1,0,1,0,0,0,0,1,1,1,0
```

And finally, consider the same LFSR, but this time seeded with 1, to produce a second version of stream b:

```
LFSR: 1,8,4,2,9,C,6,B,5,A,D,E,F,7,3
b2:   1,0,1,1,0,0,1,0,1,0,0,0,0,1,1
```

If our objective is to compute $a \times b$, we would AND stream a with stream b. If the first stream b is used, the resulting product stream is:

```
0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0
```

representing 1/16 (0.0625). On the other hand, if the second stream b is used, the resulting product stream is:

`0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0`

representing 2/16 (0.125). Evidently, the first b stream produces a result that is considerably closer to ideal, $7/16 \times 3/16 = 21/256 = 0.082$, yet the only change is in the seeding of the various LFSRs.

A. MUX-Based SC Architecture

The MUX-based stochastic computing architecture described above requires $2n + 1$ LFSRs to generate random numbers: n of these are used to generate the values of S and the other $n + 1$ are used to produce random numbers that are compared with the Bernstein coefficients. For Q bits of precision (Q-bit LFSRs), the number of ways to uniquely seed the LFSRs is $\prod_{j=0}^{2n} 2^Q - 1 - j \approx 2^{2nQ}$. Prior work does not consider how to seed such LFSRs, leading us to surmise that prior researchers seeded them arbitrarily.

We consider two approaches to seeding the LFSRs. We first note that in the MUX-based SC architecture, only the b_i's depend on the function being implemented. The circuit structure producing S is *agnostic* to the function implemented. Improvements in the accuracy of S are useful for *any* target function. Consequently, in a first approach, we consider seeding the n LFSRs in the S-structure (i.e. the portion of the circuit producing S) with the aim of improving the values of S produced to make them a better approximation to the ideal. In a second approach, we consider the impact of seeding *all* $2n + 1$ LFSRs.

We consider both approaches as practical from the hardware implementation viewpoint: to configure the MUX-based SC architecture, instead of setting solely the Bernstein coefficients as proposed in [8], one would also seed the LFSRs with specific values. Prior work assumes that the Bernstein coefficients for a given function are computed offline in software (not in the hardware itself). Likewise, we do *not* envision that the selection of seeds would be done in hardware; rather, the selection would be done offline in software and the chosen LFSR seeds would be used to configure the SC architecture, along with the Bernstein coefficients. In fact, we expect that in prior implementations, seeds are *already* being provided to configure the LFSRs. What we are proposing is to provide selected seeds, rather than arbitrarily chosen ones.

B. Seeding the S-Structure

(1) defines the probability of S being a specific value of i, given x and n. For a specific SC precision, Q, and associated bitstreams of length $l = 2^Q - 1$ (i.e. l clock cycles), ideally the following number of occurences, O, of a particular i would be observed on S:

$$O_{i,x,l} = P(S = i|x) \cdot l \tag{3}$$

The quantity $O_{i,x,l}$ is generally not an integer, so we round it to the nearest integer by taking $\lfloor O_{i,x,l}+0.5 \rfloor$, thereby establishing an "ideal" value the number of occurences of i we could expect in a practical implementation.

For a given seeding combination of the n LFSRs in the S-structure, we can set the input x to a specific value, execute the structure for l cycles and count the number of observed i's, $Obs_{i,x,l}$. The discrepancy between the observed and ideal number of occurences of a given i represents error in the output of the S structure. Let $comb_S$ be a particular seeding combination for the n seeds in the LFSRs of the S-structure, we define the error associated with $comb_S$ as:

$$Error(comb_S) = \sum_{x=0}^{2^Q-1} \sum_{i=0}^{n} (Obs_{i,x,l} - \lfloor O_{i,x,l} + 0.5 \rfloor)^2 \tag{4}$$

where the first summation walks over all possible input values x; the second summation is over the possible values of i; Q is the precision; and, $l = 2^Q - 1$ is the bitstream length. In essence, (4) is a proxy for the mismatch between the values of S observed for a specific seed combination, and those we would ideally like to see. (4) is the sum of the squared errors in the SC approximation of the actual function and is inspired by the Chi-squared test in statistics [12].

C. Seeding the Entire Structure

An alternative to tuning the seeds in the S structure is to tune all $2n + 1$ seeds in the entire circuit. For a particular seeding combination, $comb$, and precision Q, we define error as follows:

$$Error(comb) = \sum_{x=0}^{2^Q-1} (f(\frac{x}{2^Q - 1}) - \hat{f}_{Qb}(\frac{x}{2^Q - 1}))^2 \tag{5}$$

where f is the function value computed in double-precision floating point in software, and $\hat{f}_{Qb}$ is the SC result with Q-bit precision, which takes $l = 2^Q - 1$ clock cycles to compute for each value of x.

D. Overall Approach

Algorithm 1 shows the seeding approach, which uses Monte Carlo seed sweeping. The inputs to the algorithm are two sets of LFSRs: L_S being the set in the S structure and L_b being the remaining LFSRs. Lines 2-11 only apply to seeding the S structure, wherein Lines 3-9 constitute a `for` loop that iterates for $Strials$ seeding combinations of L_S. For each of these, equation (4) is evaluated and the best seeding combination (minimum error) is stored. Line 10 sets the seeds of L_S to the best combination for the remainder of the algorithm.

Lines 15-25 perform seed sweeping on a set of LFSRs, L: if the S structure is already seeded by lines 2-11, these lines sweep seeds solely on those LFSRs in L_b, i.e. $L = L_b$; otherwise, seed sweeping is performed on *all* LFSRs in the design: $L = L_b \cup L_S$. The `for` loop in lines 17-24 executes for $trials$ seeding combinations. The error for each seeding combination is computed using equation (5); average and minimum error are computed and reported in line 25.

IV. LFSR Design Alternatives

An n-bit LFSR is said to have *maximal length* if it cycles through *all* $2^n - 1$ possible states. For example, given the n-bit LFSR is seeded (initialized) to a particular n-bit state s_i ($s_i \neq 0$), the LFSR walks through all values from [1:$2^n - 1$], beginning with s_i, in pseudo-random order in $2^n - 1$ clock cycles. On the next clock cycle, the LFSR returns to the initial state s_i. The specific order through which the various states are traversed is defined by the LFSR's "feedback polynomial" [4]. The polynomial specifies the LFSR bits used in determining the next state (the taps). By seeding an LFSR differently, we change only its initial state – the order in which states are traversed is fixed for a given feedback polynomial. In this paper, we only consider the use of maximal-length LFSRs.

A. Scrambling LFSR Output Bits

Seeded differently but with uniform polynomials, the various LFSRs in the circuit are all cycling through states in the same order, but "offset" (in clock cycles) from each other by an amount equal to the distance between the seeds in the state ordering. A straightforward optimization we consider in this paper is to scramble (permute) the LFSR output bits in addition to seeding them differently. Scrambling is cheap to implement in hardware, amounting simply to a different connectivity between the LFSR outputs and the comparator inputs (c.f. Fig. 1(c)). We explore using a different randomly-selected scrambling for each LFSR. With their outputs scrambled, the LFSRs "appear" to be walking through their states in different orders. We randomly scramble the outputs of each LFSR, and then execute Algorithm 1.

Algorithm 1 Monte Carlo-style LFSR seeding approach.

input : L_S: set of LFSRs in S structure
input : L_b: set of LFSRs in b structure
output: Seed set for L_S, L_b, avg/min error

$E_{best} = \infty$
if *S-structure seeding* **then**
  for $i \leftarrow 1$ **to** $Strials$ **do**
    $comb_S$ = unique randomly chosen seeds for L_S
    if $E(comb_S) < E_{best}$ **then**
      $E_{best} = E(comb_S)$
      Store $comb_S$ as $Best_S$
    end
  end
  Restore seeds of L_S to $Best_S$
  $L = L_b$ // seed L_b below
else
  $L = L_b \cup L_S$ // seed both L_b and L_S below
end
$E_{best} = \infty$
$SumError = 0$
for $i \leftarrow 1$ **to** $trials$ **do**
  $comb$ = unique randomly chosen seeds for L
  $SumError = SumError + E(comb)$
  if $E(comb) < E_{best}$ **then**
    $E_{best} = E(comb)$
    Store $comb$ as $Best$
  end
end
Report $Best$, avg error ($SumError/trials$), min error E_{best}

B. Feedback Polynomial Diversity

A second approach we consider to improve SC accuracy (and make bitstreams more statistically independent) is to leverage the property that in general, there is more than one polynomial that produces a maximal-length LFSR [4]. For example, for 10-bit LFSRs, [5] reports at least 60 different polynomials that produce maximal-length sequences. LFSRs with different polynomials will exhibit different state orderings, potentially producing more independent SC bitstreams.

Thus, a second design optimization we consider is to implement each LFSR with a *different* polynomial. This is not quite as straightforward as scrambling outputs from the hardware implementation perspective, and it may be undesireable as it introduces irregularity to the circuitry. Nevertheless, LFSRs with different polymomials are not appreciably different from the area/speed perspective, and in fact, we have observed their speed/area to be identical in FPGA implementations. Using the polynomial tables in [4], we choose a different feedback polynomial for each LFSR arbitrarily from the table. We then execute Algorithm 1.

TABLE II. ALTERA CYCLONE IV IMPLEMENTATION RESULTS FOR VARIOUS PRECISIONS OF MUX-BASED SC.

	8-bit	9-bit	10-bit
Area (LEs)	232	258	285
Speed (MHz)	169.6	164.1	157.9
Energy/Op (nJ)	17.92	36.2	72.93

V. EXPERIMENTAL STUDY

In this paper, we use (5) to quantify the error in a SC approximation: the sum of the squared errors in the approximation. The maximum precision we consider is 10-bit (1023-bit stream lengths). To allow comparisons between the errors at different precisions, e.g. 10-bit and lower precisions (which have shorter stream lengths), we sum the error across 1024 input points, even in the lower-precision cases. For example, with 9-bit precision, we compute error as follows: $Error = \sum_{x=0}^{1023}(f(\frac{x}{1023}) - \hat{f}_{9b}(\frac{\lfloor \frac{x}{2} \rfloor}{511}))^2$. Thus, our error values for lower precisions reflect both the error from short bitstream lengths, as well as error from coarser quantization.

For hardware implementation, we implemented the MUX-based SC architecture at several precisions in Verilog RTL. We synthesize the Verilog to the Altera Cyclone IV $45nm$ FPGA [2]. Energy results are based on post-place-and-route delay and capacitance extraction. Circuits were simulated using ModelSim with full routing delays, producing switching activity in VCD format. The activity and capacitance data is used by Altera's PowerPlay power analyzer tool. The energy results reflect dynamic power consumption consumed in the FPGA core logic (i.e. not including I/O power or leakage).

Table II shows area, speed, and energy consumption for the MUX-based SC architecture, 6^{th}-order, at various precisions, for the function $y = x^{0.45}$. Area is given in Cyclone IV logic elements (LEs), each of which comprises a 4-input look-up-table (LUT) paired with a flip-flop. Delving into the area numbers, the 9-bit and 10-bit implementations are $\sim$11% and $\sim$22% larger than the 8-bit implementation, respectively. This agrees with intuition, as for example, the register widths in the 10-bit version are 25% wider than those in the 8-bit version. Regarding circuit speed, we observe a degradation as precision is increased, likely owing to wider comparators. The energy numbers in the table represent the total energy to compute one output value for an input value: this takes 255, 511, and 1023 cycles in the 8-, 9-, and 10-bit precision cases. As expected, energy consumption roughly doubles for each bit increase in precision. Meaning, computing a value in 10-bit precisions requires $4\times$ the energy of 8-bit precision.

Turning now to seed sweeping, Table I shows the impact of seed selection for a set of five benchmarks, listed in column 1. The results are for 6^{th}-order Bernstein approximations with 1023-bit stream lengths. Column 2 shows the average error across 1000 different seedings of *all* 13 LFSRs in the MUX-based stochastic computing circuit (Fig. 2). We refer to the results in this column as the *baseline* in the remainder of this paper. The numbers represent the average error one could expect with arbitrarily chosen seeds. Column 3 of the table shows the minimum error achieved for *one* of the seeding combinations; i.e. the best error achieved among the 1000 seeding trials. The second-last row of the table shows geometric mean results; the last row shows the ratio of geomean relative to the baseline. Observe that, on average, the minimum error achieved is just 20% of the average error:

TABLE I. AVERAGE AND MIN ERROR RESULTS FOR BASELINE, SCRAMBLING AND MULTI-POLY BITSTREAM GENERATION DESIGNS (1000 TRIALS).

	Baseline		Scrambling		Multi-Poly	
Benchmark	**Average error**	**Min error**	**Average error**	**Min error**	**Average error**	**Min error**
pow(x,0.45)	0.153	0.042	0.146	0.051	0.145	0.050
exp(-3x)	0.097	0.016	0.095	0.017	0.093	0.019
tanh(2x)	0.091	0.017	0.089	0.017	0.092	0.024
(sin(4x)+2)/4	0.141	0.029	0.135	0.029	0.141	0.028
(1 + 0.3*cos(5x))sin(x)	0.150	0.028	0.137	0.029	0.137	0.033
Geomean	**0.123**	**0.025**	**0.118**	**0.026**	**0.119**	**0.029**
Ratio	**1.00**	**0.20**	**0.95**	**0.21**	**0.97**	**0.24**

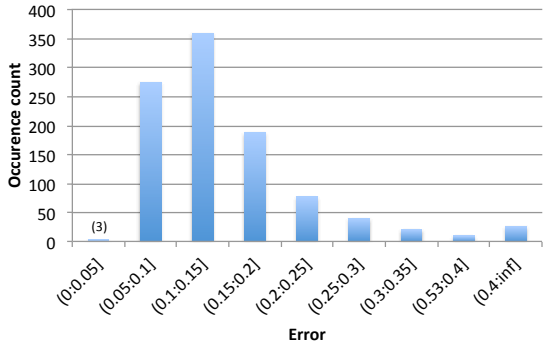

Fig. 3. Error histogram for benchmark $y = x^{0.45}$ across 1000 seeding trials.

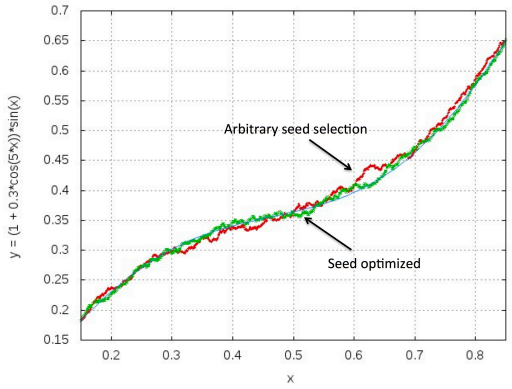

Fig. 4. Arbitrary and optimized seeding for a function.

$5\times$ lower than the average. The results show clearly that some seeding combinations yield considerably lower errors than other combinations.

To help understand the relative scarcity of "good" seeding combinations versus average ones, Fig. 3 shows a histogram of errors for one of the benchmarks, $y = x^{0.45}$. Most of the errors are clustered around the mean value of 0.153; just 3 seeding combinations had errors less than 0.05. This suggests that with arbitrary seed selection, one is not likely to land on a good combination. Fig. 4 shows SC results for: 1) an arbitrary LFSR seeding (red points), and 2) an optimized seeding (green points) for one of the benchmarks. A "best fit" line is also shown, corresponding to double-precision floating point. Qualitatively, it is apparent the optimized seeding produces results that remain far closer to the best-fit line.

Columns 4 and 5 of Table I show results for the design incorporating LFSR output bit scrambling. Columns 6 and 7 of the table show results corresponding to the use of a different feedback polynomial for each LFSR. Both design alternatives have a modest impact on the average error versus the baseline, reducing it by 5% and 3% for scrambling and multiple polynomials, respectively. However, observe that for both alternatives, the minimum error (columns 5 and 7) is not reduced (and actually slightly increased) relative to the minimum error for the baseline, due likely to "noise" in the seed-sampling process, given the huge search space and the fact that the minimum error is for a single seeding combination.

TABLE III. RESULTS FOR S-STRUCTURE SEEDING (1000 TRIALS).

	No Scrambling		Scrambling	
Benchmark	**Avg. error**	**Min error**	**Avg. error**	**Min error**
pow(x,0.45)	0.131	0.038	0.118	0.039
exp(-3x)	0.085	0.023	0.088	0.018
tanh(2x)	0.088	0.018	0.092	0.022
(sin(4x)+2)/4	0.136	0.029	0.136	0.025
(1 + 0.3*cos(5x))sin(x)	0.133	0.024	0.128	0.030
Geomean	0.112	0.025	0.111	0.026
Ratio vs. baseline	0.91	0.21	0.90	0.21

Table III shows results for S-structure seeding (see Section III-B). The results are again for 1000 seeding trials; however, the seeds in the S-structure are fixed in advance based on optimizing (4) across 50×10^6 trials (c.f. Algorithm 1: $Strials = 50 \times 10^6$, $trials = 1000$). The last row of the table shows there is a benefit in seeding the S-structure. Columns 2 and 3 of the table show results for S-structure seeding without scrambling or the use of different feedback polynomials. Average error is reduced by 9% relative to the baseline (last row of table). The minimum error is about the same as the baseline. Columns 4 and 5 of the table show results for S-structure seeding *combined* with scrambling (combining S-structure seeding with multiple feedback polynomials produced similar results). Observe that average error is reduced by 10% vs. the baseline, nearly the same as when scrambling is not applied. While S-structure seeding offers promise, it is clear that the "best" seeding combination for a given target function (considering seeding *all* 13 LFSRs simultaneously) produces superior results vs. the average results for a well-tuned S-structure.

The minimum error results in column 3 of Table I reflect a specific seeding combination among 1000 trials for each circuit. A natural question is whether a good seeding combination for one circuit is also a good combination for another circuit. Table IV shows results for *cross-seeding*: finding the best seeding combination for a function, and evaluating that combination across all other functions. The rows of the table correspond to the functions we used to select the best combination; the columns correspond to the evaluation functions. The diagonals show the minimum-error data in column 3 of Table I. Observe that the non-diagonal data values in each column of Table IV are often lower than the average for the corresponding function in column 2 of Table I. However, in general, the values are significantly higher than the minimum error observed for that function (in column 3 of Table I). From this, we conclude that while there may be some benefit in sharing seeding combinations across different functions, far better results are achieved if a unique "good" combination is

TABLE IV. EVALUATING THE IMPACT OF "CROSS-SEEDING" WHERE THE BEST SEEDS FOR EACH FUNCTION (ROWS) ARE USED ACROSS ALL FUNCTIONS (COLUMNS).

	Evaluation function				
Training function	pow(x,0.45)	exp(-3x)	tanh(2x)	(sin(4x)+2)/4	(1 + 0.3*cos(5x))sin(x)
pow(x,0.45)	0.042	0.038	0.047	0.050	0.140
exp(-3x)	0.149	0.016	0.028	0.088	0.129
tanh(2x)	0.568	0.034	0.017	0.579	0.131
(sin(4x)+2)/4	0.096	0.043	0.097	0.029	0.092
(1 + 0.3*cos(5x))sin(x)	0.091	0.052	0.054	0.115	0.028

TABLE V. COMPARISON ACROSS 10-BIT, 9-BIT AND 8-BIT PRECISION (1023-BIT, 511-BIT, 255-BIT STREAM LENGTHS, RESPECTIVELY).

	Baseline 10-bit precision		**9-bit precision**		**8-bit precision**	
Benchmark	**Average error**	**Min error**	**Average error**	**Min error**	**Average error**	**Min error**
pow(x,0.45)	0.153	0.042	0.295	0.075	0.606	0.153
exp(-3x)	0.097	0.016	0.188	0.033	0.381	0.081
tanh(2x)	0.091	0.017	0.198	0.030	0.420	0.110
(sin(4x)+2)/4	0.141	0.029	0.286	0.062	0.579	0.140
(1 + 0.3*cos(5x))sin(x)	0.150	0.028	0.300	0.068	0.616	0.131
Geomean	**0.123**	**0.025**	**0.248**	**0.050**	**0.510**	**0.120**
Ratio	**1.00**	**0.20**	**2.01**	**0.41**	**4.14**	**0.97**

selected for each function.

Table V shows results for lower precisions, and repeats the data for 10-bit precision for convenience. The last row of columns 2, 4 and 6 shows average error data across 1000 seeding trials. Observe that error is, on average, $2\times$ higher when 9-bit precision is used vs. 10-bit precision, and is $4.1\times$ higher when 8-bit precision is used. As mentioned above, the increased error reflects both shorter stream lengths, and coarser quantization. Turning to the minimum error data in columns 3, 5 and 7, we observe that with 9-bit precision, the minimum error observed is 59% lower, on average, than the *average* error for 10-bit precision. This implies that by using bitstreams that are *half* as long (i.e. 511 bits vs. 1023 bits), with seed selection, 9-bit precision can deliver higher accuracy than 10-bit precision with arbitrarily chosen seeds. Surprisingly, with 8-bit precision, the minimum error observed is 3% lower than the average error with 10-bit precision. There are clearly significant error-reduction benefits in careful seeding of LFSRs. In combination with the energy results presented earlier, the accuracy results above demonstrate that SC with careful seeding in 8-bit precision yields nearly the same accuracy as SC in 10-bit precision seeded arbitrarily – a $4\times$ energy reduction for a given accuracy level.

Seed selection thus offers two benefits: 1) raising the accuracy of SC for a given precision level (e.g. 10-bit, 9-bit or 8-bit), or 2) permitting lower precisions to be used for the same or better accuracy vs. with arbitrarily chosen seeds. For the latter, consider that 8-bit SC uses 255-bit stream lengths, which are $4\times$ shorter than the 1023-bit stream lengths needed for 10-bit accuracy. This represents a $4\times$ delay improvement.

VI. CONCLUSIONS AND FUTURE WORK

We considered the design of bitstream generators for use in stochastic computing (SC). Two LFSR design alternatives were investigated to raise diversity in the state sequence: scrambling LFSR outputs, and the use multiple feedback polynomials. Both approaches demonstrate reduced error in SC results. We also studied the impact of LFSR seed sweeping on SC accuracy. Experimental results show that one can reap considerable accuracy improvements from LFSR seeding, with a key result being that careful seeding of 8-bit precision SC can provide higher accuracy than arbitrary seeding of 10-bit precision SC. Analysis of an FPGA implementation of SC structures showed that $\sim 2\times$ more energy is needed for each bit of increased SC precision. Hence, seed sweeping permits the use of lower-precision SC for a given accuracy level, yielding significant energy reductions. As future work, we plan to explore the benefit of using n-bit LFSRs/comparators with stream lengths shorter than $2^n - 1$ bits, which may reducing quantization error at modest hardware cost.

VII. ACKNOWLEDGEMENTS

The authors thank Weikang Qian (Shanghai Jiao Tong University) for providing the MATLAB code to find the Bernstein coefficients, and George Constantinides and David Thomas (Imperial College) for their helpful comments on this work. The research was partly funded by the Japan Society for the Promotion of Science Visiting Fellow Program and KAKENHI 15H02679.

REFERENCES

[1] A. Alaghi and J. P. Hayes. Survey of stochastic computing. *ACM Trans. Embed. Comput. Syst.*, 12(2s):92:1–92:19, 2013.

[2] Altera, Corp. *Cyclone-IV FPGA family datasheet*, 2015.

[3] C. Fagot, O. Gascuel, P. Girard, and C. Landrault. On calculating efficient LFSR seeds for built-in self test. In *European Test Workshop*, pages 7–14, 1999.

[4] A. Klein. *Sream Ciphers (Chapter 2)*. Springer-Verlag, 2013.

[5] P. Koopman. Maximal length LFSR feedback terms, 2015. `http://users.ece.cmu.edu/ koopman/lfsr/index.html`.

[6] P. Li and D. Lilja. Using stochastic computing to implement digital image processing algorithms. In *IEEE ICCD*, pages 154–161, 2011.

[7] G. Lorenz. *Bernstein Polynomials*. AMS Chelsea Publishing, New York, NY, 1986.

[8] W. Qian, X. Li, M. Riedel, K. Bazargan, and D. Lilja. An architecture for fault-tolerant computation with stochastic logic. *IEEE Trans. on Computers*, 60(1):93–105, 2011.

[9] W. Qian, M. Riedel, K. Bazargan, and D. Lilja. The synthesis of combinational logic to generate probabilities. In *IEEE/ACM ICCAD*, pages 367–374, 2009.

[10] W. Qian, C. Wang, P. Li, D. Lilja, K. Bazargan, and M. Riedel. An efficient implementation of numerical integration using logical computation on stochastic bit streams. In *IEEE/ACM ICCAD*, pages 156–162, 2012.

[11] N. Saraf and K. Bazargan. Sequential logic to transform probabilities. In *IEEE/ACM ICCAD*, pages 732–738, 2013.

[12] R. Walpole, R. Myers, S. Myers, and K. Ye. *Probability and Statistics for Engineers and Scientists*. Pearson, Toronto, ON, 2010.

[13] Z. Wang, K. Chakrabarty, and M. Bienek. A seed-selection method to increase defect coverage for LFSR-reseeding-based test compression. In *IEEE European Test Symp.*, pages 125–130, 2007.