An Efficient Implementation of Numerical Integration Using Logical Computation on Stochastic Bit Streams [Special Session Paper]

Weikang Qian and Chen Wang
University of Michigan-SJTU Joint Institute
Shanghai Jiao Tong University
Shanghai, China
{qianwk, wangchen_2007}@sjtu.edu.cn

Peng Li, David J. Lilja, Kia Bazargan, and Marc D. Riedel
ECE Department, University of Minnesota
Minneapolis, MN, USA
{lipeng, lilja, kia, mriedel}@umn.edu

ABSTRACT

Numerical integration is a widely used approach for computing an approximate result of a definite integral. Conventional digital implementations of numerical integration using binary radix encoding are costly in terms of hardware and have long computational delay. This work proposes a novel method for performing numerical integration based on the paradigm of logical computation on stochastic bit streams. In this paradigm, ordinary digital circuits are employed but they operate on stochastic bit streams instead of deterministic values; the signal value is encoded by the probability of obtaining a one versus a zero in the streams. With this type of computation, complex arithmetic operations can be implemented with very simple circuitry. However, typically, such stochastic implementations have long computational delay, since long bit streams are required to encode precise values. This paper proposes a stochastic design for numerical integration characterized by both small area and short delay – so, in contrast to previous applications, a win on both metrics. The design is based on mathematical analysis that demonstrates that the summation of a large number of terms in the numerical integration could lead to a significant delay reduction. An architecture is proposed for this task. Experiments confirm that the stochastic implementation has smaller area and shorter delay than conventional implementations.

1. INTRODUCTION

Numerical integration is a widely used approach for computing an *approximate* solution to a definite integral $\int_a^b g(x)dx$ [1]. It is applied in situations where it is difficult or impossible to find an antiderivative of the integrand, for example, in the case where the integrand is e^{-x^2}. A basic form of numerical integration is to approximate the integral as

$$\int_a^b g(x)dx \approx \frac{b-a}{M} \sum_{i=0}^{M-1} g\left(a + \frac{i(b-a)}{M}\right) \quad (1)$$

Conventional digital implementations of numerical integration are costly in terms of hardware, since they employ complex arithmetic circuits for calculating the function g and an adder for accumulating $g(x)$'s for different x points. Conventional implementations also have high delay. Since we need to calculate M $g(x)$'s and then sum them together, according to Equation (1), the time consumption is MT_C, where T_C is the time required for calculating a single $g(x)$.

In this work, we propose a novel method for performing numerical integration based on the paradigm of logical computation on stochastic bit streams. In this paradigm, ordinary digital circuits are employed but they operate on stochastic bit streams instead of deterministic values; the signal value is encoded by the probability of obtaining a one versus a zero in the streams. With this type of computation, complex arithmetic operations can be implemented with very simple circuitry. The major drawback is that lengthy bit streams are needed to encode precise values and hence there is a typically a long computational delay. This drawback is due to the error and the precision issues. Stochastic encoding is subject to errors caused by its inherent randomness. We need to increase the bit length to decrease the error below a threshold. Also, the precision of a value encoded by stochastic bit stream is proportional to the length of the stream. We need to increase the bit length to achieve a high precision.

Through mathematical analysis, this paper demonstrates that when performing numerical integration, the summation of a large number of terms can be performed with comparatively *low delay* in stochastic implementations. Our contributions in this paper are: 1) we provide a theoretical analysis showing that the delay of the stochastic implementation of numerical integration can be significantly reduced while achieving a small error bound and a high precision; and 2) we propose an architecture for the stochastic implementation of numerical integration. Experimental results show that our stochastic implementation has both a smaller circuit area and a shorter delay than the conventional implementation using binary radix encoding.

The remainder of this paper is organized as follows. Section 2 introduces the background on logical computation on stochastic bit streams and points to some related works. Section 3 presents a theoretical analysis on the delay reduction with the stochastic implementation. Section 4 shows the architecture for the stochastic implementation of numerical integration. Section 5 presents experimental results. Finally, Section 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

Traditional arithmetic circuits operate on numbers encoded by binary radix, which is a deterministic way to represent numerical values with zeros and ones. Fundamentally different from the binary radix, stochastic encoding is another way to represent numerical values by logical zeros and ones [2, 3]. In such a encoding, a real value p in the unit interval is represented by a sequence of N *random* bits $X_1, X_2, \ldots, X_N \in \{0, 1\}$, with each X_i having probability p of being one and probability $(1 - p)$ of being zero, i.e.,

IEEE/ACM International Conference on Computer-Aided Design (ICCAD) 2012, November 5-8, 2012, San Jose, California, USA.

$P(X_i = 1) = p$ and $P(X_i = 0) = 1 - p$. Typically, a sequence of random bits is generated serially in time to form a stochastic bit *stream*. Figure 1(a) shows an example of a stochastic bit stream encoding the value $5/8$.

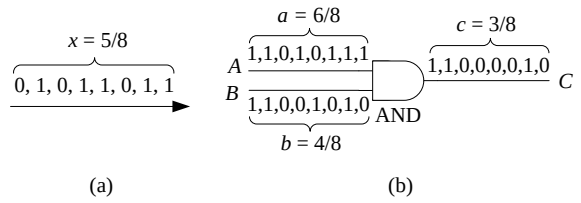

Figure 1: Stochastic encoding and computation on stochastic encoding: (a) A stochastic bit stream encoding the value $x = 5/8$; (b) An AND gate multiplying two values encoded by two input stochastic bit streams.

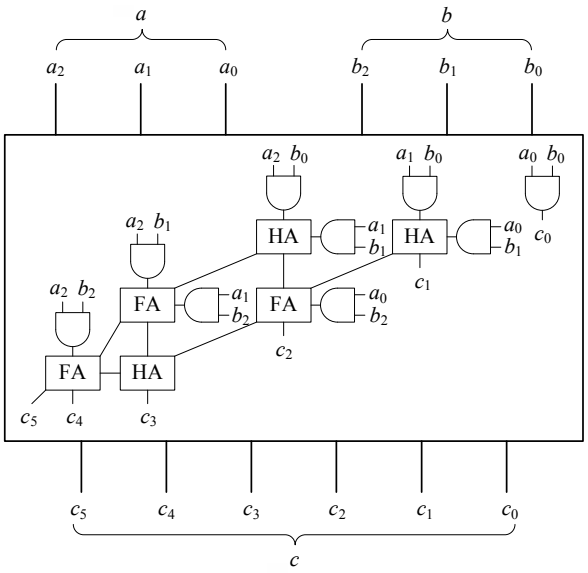

Figure 2: Multiplication using conventional binary radix encoding: a carry-save multiplier, operating on 3-bit binary radix encoded inputs A and B. "FA" refers to a full adder and "HA" refers to a half adder.

Since the random sequences are composed of binary digits, we can apply digital circuits to process them. Thus, instead of mapping Boolean values into Boolean values, a digital circuit now maps real probability values into real probability values. We refer to this type of computation as *logical computation on stochastic bit streams*.

Computation on stochastic bit streams can reduce hardware cost significantly. For example, multiplication can be implemented with a single AND gate. As we know, an AND gate outputs a logical one if and only if both of its inputs are one. Now if the two inputs are *independent* stochastic bit streams, then the probability of obtaining a one in the output bit stream equals the *product* of the probabilities of obtaining a one in the input streams. Therefore, an AND gate multiplies the values encoded by stochastic bit streams. Figure 1(b) illustrates an AND gate performing multiplication on stochastic bit streams. In contrast, conventional multiplier is very hardware-consuming. Figure 2 shows a conventional design for a 3-bit carry-save multiplier, operating on binary radix-encoded numbers. It consists of 9 AND gates, 3 half adders, and 3 full adders, for a total of 30 gates.[1] Besides multiplication, other basic arithmetic operations such as addition, division, and square root can also be implemented with very simple digital constructs using stochastic encoding [3, 4].

However, the major drawback of stochastic encoding is that it typically needs a long bit stream to encode a value. One reason is because stochastic encoding suffers from errors due to its stochastic nature and we need to increase the bit length to reduce the error. Mathematically, a stochastic bit stream we observe, such as the one in Figure 1(a), is just a *trial* from a *Bernoulli process* $X_1, X_2, \ldots, X_N$, where each X_i is a *Bernoulli random variable*, taking value 1 with probability p and value 0 with probability $1-p$. We obtain the value represented by a stochastic bit stream by counting the number of ones in that stream and then dividing it by the total length N. However, this value does not necessarily equal to p. Indeed, the value represented by a stochastic bit stream is just a sample from the random variable

$$Y = \frac{1}{N}(X_1 + \cdots + X_N),$$

which is a *binomial random variable* taking values from the set $\{0, \frac{1}{N}, \ldots, \frac{N-1}{N}, 1\}$. Although the expectation of Y is

$$\mathrm{E}[Y] = \frac{1}{N}(\mathrm{E}[X_1] + \cdots + \mathrm{E}[X_N]) = p,$$

a sample of Y, which is based on a trial of the underlying Bernoulli process, does not necessarily equal p. Such a difference between the observed value and the actual value p is due to the stochastic nature of this encoding.

The way to reduce the error due to randomness is to increase the length of the bit stream. Since $p = \mathrm{E}[Y]$, the *expected* difference between a sample of Y and p equals the standard deviation of Y, which can be calculated as [2]

$$\sigma_Y = \sqrt{\mathrm{Var}[Y]} = \sqrt{\frac{p(1-p)}{N}}.$$

If we require the expected error to be less than a threshold ϵ, then we have

$$\sqrt{\frac{p(1-p)}{N}} < \epsilon.$$

This requires that the bit length N is larger than $\frac{p(1-p)}{\epsilon^2}$. Thus, from the error aspect, stochastic encoding requires a long bit stream.

Besides the reason due to error, from the precision aspect, we also require a long bit stream for stochastic encoding. By its nature, stochastic encoding is a uniform encoding with each bit contributing the same weight to the encoded value. Thus, to represent a value with precision $\frac{1}{2^n}$, the length of the stochastic bit stream should be at least 2^n. This is much longer than the bit length of binary radix encoding, which requires only n bits to achieve the same precision.

Since the stochastic implementation processes one bit per clock cycle and its bit length is prohibitively long, logical computation on stochastic bit stream is very time-consuming. Without taking stochastic error into account, if we require a precision of $\frac{1}{2^n}$, then we need at least 2^n clock cycles to obtain the result.

The drawback of long delays could be partially alleviated by reducing the basic clock period due to the simplicity of the circuitry. In the example of stochastic multiplier, the clock period can be as short as the delay of a single AND gate, which is much less than the critical path delay of a conventional multiplier. Assume that the critical path delay of the stochastic implementation and that of the conventional implementation are T_S and T_C, respectively. However, even if T_S is much smaller than T_C, the computation delay overhead of stochastic implementation compared to conventional implementation is still

$$2^n \frac{T_S}{T_C}, \tag{2}$$

[1] A half adder can be implemented with one XOR gate and one AND gate. A full adder can be implemented with two XOR gates, two AND gates, and one OR gate.

which could be very large for a large n.

The issue of long delays can be mitigated through parallel processing. Instead of using a bit stream of length N to represent a value, we can use L bit streams of length $\frac{N}{L}$ to represent the same value. Figure 3 shows how we can represent the value $5/8$ by two stochastic bit streams of length 4. If we split a stochastic bit stream into L shorter streams, the computation delay reduces to $\frac{1}{L}$ of the original one. However, as a trade-off, we need L copies of the original circuit to process L bit streams in parallel. For example, as shown in Figure 3, if we want to multiply two values, each encoded by two shorter bit streams, we need two AND gates. Thus, although parallel processing could reduce the computation delay to $1/L$ of the original one, it increases the circuit area by L times. This solution is not decent. Further, when circuit area is constrained, this solution is not viable.

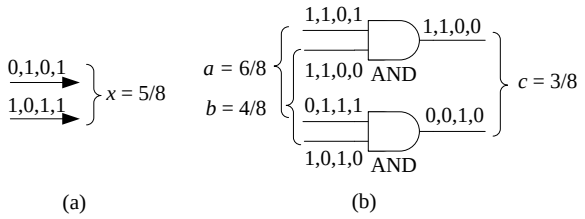

Figure 3: Parallel implementation of logical computation on stochastic bit streams. (a) Representing a value by two shorter stochastic bit streams. (b) Multiplication on values encoded as two shorter stochastic bit streams.

Although the paradigm of logical computation on stochastic bit streams suffers from long delays, it still finds applications in areas such as artificial neural networks (ANNs), communication, and image processing [5–10]. It is a particularly good fit for ANNs since applications in this area typically have a large number of multipliers and adders [5–8]. Due to the simplicity of the stochastic multiplier and adders, researchers are able to build large scale ANNs using these constructs. Recently, the paradigm has been used to implement a low-density parity-checking (LDPC) decoder, a construct widely used in communication for error correction [9]. It has also been applied in image processing in implementing functions such as edge detection, median filter, and contrast adjusting [10].

All of the applications proposed so far only take advantage of the small circuit area that logical computation on stochastic bit streams provides; they all suffer from long delays. In this work, we apply the paradigm to implement numerical integration. The result is a design with both a smaller area and a shorter computation delay than the conventional implementations using binary radix – so, in contrast to previous applications, a win on both metrics.

3. REDUCING COMPUTATION DELAY OF NUMERICAL INTEGRATION

We intend to use logical computation on stochastic bit streams to calculate the numerical integration formula shown in Equation (1). Since this type of computation requires both the inputs and the outputs to be a probability value in the unit interval, we need to do some pre-processing to transform the integration interval into the unit interval and the original function $g(x)$ into a function $f(x)$ that maps the unit interval onto itself, i.e., $f(x) \in [0,1]$ for any $x \in [0,1]$. This can be achieved by applying affine transformations. Now we assume that we are integrating a function $f(x)$ in the unit interval $[0,1]$ and $f(x)$ maps the unit interval into itself. The numerical integration formula becomes

$$\frac{1}{M}\sum_{i=0}^{M-1} f\left(\frac{i}{M}\right) \tag{3}$$

The stochastic implementation is based on a circuit that maps an input probability value x into an output probability value $f(x)$, which will be discussed in the next section. We use this circuit to compute $f(\frac{i}{M})$ for each $i = 0, \ldots, M-1$ and then average these values. If we are only interested in evaluating $f(x)$ on a single point, then we need a long bit stream for the stochastic implementation due to the error and the precision issues. However, in numerical integration where multiple evaluations of $f(x)$ are finally averaged, we can significantly reduce the length of the bit stream for representing each $f(\frac{i}{M})$, and hence, reduce the computation delay.

We consider an extreme case where each $f(\frac{i}{M})$ is represented by just one bit. Then, the circuit that computes $f(x)$ stochastically will operate for only one clock cycle in calculating $f(\frac{i}{M})$. Thus, its output is a single sample from a Bernoulli random variable Y_i that takes value 1 with probability $f(x_i)$ and value 0 with probability $1 - f(x_i)$. This sample, which can only be a 0 or a 1, is much away from the probability value $f(x_i)$ due to a limited number of samplings. However, the numerical integration involves averaging over these samples and the error of each sample is stochastic by its nature. Thus, although the error is large for each sample, the averaging of the samples may cancel out these errors. Mathematically, since the i-th bit ($i = 0, \ldots, M-1$) we get is a sample from a Bernoulli random variable Y_i with $P(Y_i = 1) = f(\frac{i}{M})$ and we eventually average over these bits, the final output is a sample from a random variable

$$Y = \frac{1}{M}(Y_0 + \ldots + Y_{M-1}).$$

Since $\mathrm{E}[Y_i] = f(\frac{i}{M})$, we can obtain the mean of Y as

$$\mathrm{E}[Y] = \frac{1}{M}\sum_{i=0}^{M-1}\mathrm{E}[Y_i] = \frac{1}{M}\sum_{i=0}^{M-1} f\left(\frac{i}{M}\right),$$

which equals Equation (3). Thus, the stochastic implementation with just one bit to represent each probability $f(\frac{i}{M})$ is an unbiased estimation. Further, since $\mathrm{E}[Y]$ equals the ideal numerical integration value, the *expected* difference between a sample of Y and the ideal value equals the standard deviation of Y. By our implementation, all the random variables Y_i are independent. Thus, the standard deviation of Y can be calculated as

$$\sigma_Y = \sqrt{\mathrm{Var}[Y]} = \sqrt{\mathrm{Var}\left[\frac{1}{M}\sum_{i=0}^{M-1} Y_i\right]} = \sqrt{\frac{1}{M^2}\sum_{i=0}^{M-1}\mathrm{Var}[Y_i]}.$$

Since Y_i is a Bernoulli random variable with probability $f(\frac{i}{M})$ of being one, its variance is

$$\mathrm{Var}[Y_i] = f\left(\frac{i}{M}\right)\left(1 - f\left(\frac{i}{M}\right)\right) \le \frac{1}{4}.$$

Therefore, we have

$$\sigma_Y \le \frac{1}{2\sqrt{M}}.$$

We can see that although we only use one bit to represent each probability value $f(\frac{i}{M})$, the expected error after averaging is well bounded. It is small if the numerical integration consists of many function evaluations. Also, since the final value is of the form $\frac{k}{M}$, where k is the total number of ones among all the bits, the precision of the computation is $\frac{1}{M}$, which is small given a large M.

The above analysis can be generalized if we use a stochastic bit stream of length L to encode each probability value $f(\frac{i}{M})$, i.e., the circuit operates L clock cycles to obtain a stochastic encoding for the value $f(\frac{i}{M})$. In this general situation, it can be shown that the

final result is a sample from a random variance Y with mean

$$\mathrm{E}[Y] = \frac{1}{M}\sum_{i=0}^{M-1} f\left(\frac{i}{M}\right),$$

and standard deviation

$$\sigma_Y \leq \frac{1}{2\sqrt{LM}}. \tag{4}$$

This means that increasing the bit length L does not affect the mean, but it decreases the expected error. Further, the precision of the computation is $\frac{1}{LM}$, which decreases by increasing L.

Finally, we analyze the computation delay. Suppose that the delay of generating a single output bit by the stochastic implementation is T_S and that the delay of evaluating a single point $f(\frac{i}{M})$ by the conventional implementation is T_C. In obtaining the final integration result, the stochastic implementation needs to evaluate M integration points, with each evaluation encoded as a bit stream of length L. Thus, the total time for obtaining the integration result is $t_S = LMT_S$. For the conventional implementation, since it also needs to evaluate M points, the entire time is $t_C = MT_C$. Thus, the delay overhead of the stochastic implementation compared to the conventional implementation is

$$\frac{t_S}{t_C} = L\frac{T_S}{T_C}.$$

As we point out in Section 2, due to the simplicity of the circuit that computes on stochastic bit streams, T_S is smaller than T_C. Then, in the extreme case where $L = 1$, the delay of the stochastic implementation is smaller than that of the conventional implementation. If we want to achieve a precision of $\frac{1}{2^n}$, we require $LM = 2^n$, then the delay overhead becomes

$$\frac{t_S}{t_C} = L\frac{T_S}{T_C} = \frac{2^n}{M}\cdot\frac{T_S}{T_C}. \tag{5}$$

Therefore, if M is large and n is moderate, the delay of stochastic implementation still could be less than that of the conventional implementation. Furthermore, comparing Equation (5) with (2), we can see that for numerical integration which involves an "averaging" of M values, the delay overhead of its stochastic implementation $\frac{t_S}{t_C}$ will be only $\frac{1}{M}$ of that for an application without the "averaging."

4. ARCHITECTURE FOR THE STOCHASTIC IMPLEMENTATION OF NUMERICAL INTEGRATION

In this section, we present the architecture for the stochastic implementation of the numerical integration. The system is shown in Figure 4, which includes three parts: the stochastic computing unit, the stochastic sweeping unit, and the de-randomizer. The system requires some independent and uniformly distributed random numbers. We assume that these random numbers are provided from external random sources. For example, linear feedback shift register (LFSR) can be used to generate these random numbers.

4.1 Stochastic Computing Unit

The stochastic computing unit (SCU) is the computing core of the entire system, which implements the integrand $f(x)$ stochastically. The SCU is modified from the circuit we proposed before which can implement an arbitrary arithmetic function [11]. The specific example shown in Figure 4 implements a *Bernstein polynomial* of degree 3. In general, a Bernstein polynomial of degree d is of the form [12]

$$B_d(x) = \sum_{i=0}^{d} b_{i,d}B_{i,d}(x),$$

where each $b_{i,d}$ is a real constant and each $B_{i,d}(x)$ is a *Bernstein basis polynomial* of the form

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

The SCU consists of a d-input adder, a $(d+1)$-to-1 multiplexers with channel bit width n, and a n-bit comparator. For the example in Figure 4, $d = 3$ and $n = 10$. The functionality of the SCU is to generate a random bit with probability $B_d(x)$ of being one, where $B_d(\cdot)$ could be an arbitrary user-specified Bernstein polynomial with all the coefficients in the unit interval and x is an evaluation point. In order to achieve this, the adder takes d inputs $X_1, \ldots, X_d$, each being an *independent* random bit with probability x of being one. The adder outputs the sum of the d random input bits. The sum is encoded by the binary radix and could be any value from the set $\{0, 1, \ldots, d\}$. The multiplexer takes the output of the adder as its selection input and $C_0[n-1:0], \ldots, C_d[n-1:0]$ as its data inputs. For the entire duration in computing an integral, we hold the data inputs $C_0, \ldots, C_d$, which are determined by the integrand. Note that each channel of the multiplexer is of bit width n. If the adder output is k ($0 \leq k \leq d$), the multiplexer will choose C_k as its output. Finally, the comparator compares the output of the multiplexer with a random number $R[n-1:0]$, which is generated by an external random source and is uniformaly distributed in the set $\{0, 1, \ldots, 2^n - 1\}$. If $R < C_k$, the output of the comparator is 1; otherwise, it is 0. Thus, the final output bit Y of the SCU is a Bernoulli random variable. The probability of Y to be one is determined by both the probability that R is less than C_k and the probability that the output of the adder is k, i.e.,

$$\begin{aligned} P(Y=1) &= \sum_{k=0}^{d} P\left(Y=1 \Big| \sum_{i=1}^{d} X_i = k\right) P\left(\sum_{i=1}^{d} X_i = k\right) \\ &= \sum_{k=0}^{d} P\left(R < C_k\right) P\left(\sum_{i=1}^{d} X_i = k\right) \end{aligned}$$

Given that $X_1, \ldots, X_d$ are independent Bernoulli random variables taking value 1 with probability x, $\sum_{i=1}^{d} X_i$ is a binomial random variable taking value in the set $\{0, \ldots, d\}$, and for $k = 0, \ldots, d$,

$$P\left(\sum_{i=1}^{d} X_i = k\right) = \binom{d}{k}x^k(1-x)^{d-k}.$$

Further, since R is a random number uniformly distributed in the set $\{0, 1, \ldots, 2^n - 1\}$, $P(R < C_k)$ is $\frac{C_k}{2^n}$. Thus, if we set C_k to be $2^n b_{k,d}$ with $0 \leq b_{k,d} \leq 1$, then the probability of Y to be one is

$$P(Y=1) = \sum_{k=0}^{d} b_{k,d}B_{k,d}(x) = B_d(x).$$

Thus, the SCU transforms random bits with probability x of being one into a random bit with probability $B_d(x)$ of being one. As we can see, we can implement arbitrary Bernstein polynomial with coefficients in the unit interval by configuring the values C_k. Taking this advantage of reconfigurability, we can implement an arbitrary arithmetic function by approximating it with a Bernstein polynomial with coefficients in the unit interval. To find a good approximation, we can formulate and solve an optimization problem using the method we proposed in [11].

4.2 Stochastic Sweeping Unit and De-Randomizer

The stochastic sweeping unit (SWU) provides the random bits to the X inputs of the SCU, as shown in Figure 4. It generates random bits with probabilities sweeping from 0 to $\frac{M-1}{M}$ with a

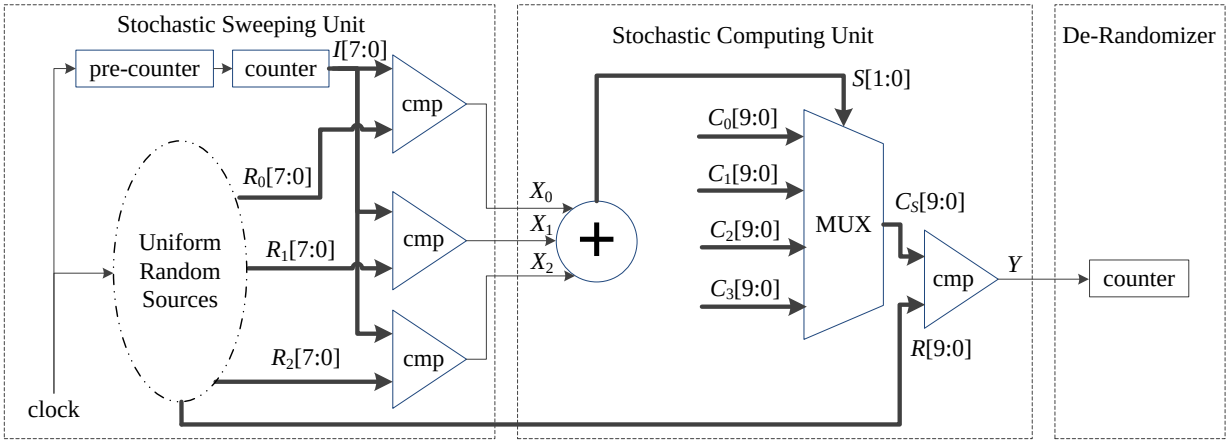

Figure 4: The architecture for the stochastic implementation of the numerical integration. "cmp" refers to a digital comparator. The thick lines are buses and their width is denoted within its signal name. For example, $R[9:0]$ refers to a bus carrying 10 bits.

step size of $\frac{1}{M}$. Hence, we call it stochastic sweeping unit. For the sake of simple implementation, we choose $M = 2^m$ where m is an integer. The SWU consists of a pre-counter, a m-bit counter, and d comparators, where d is the degree of the Bernstein polynomial implemented by the SCU.

The pre-counter counts from 0 to $L-1$ cyclically, where L is the number of bits to encode a specific probability value. Also, for the sake of simple implementation, we choose $L = 2^l$ where l is an integer. Each time the pre-counter reaches its maximal value $L-1$, it increments the counter by 1.

The counter counts from 0 to $M-1$. When the value of the counter is k, the outputs of the d comparators are all random bits with probability $\frac{k}{M}$ of being one. This is achieved by comparing k with d uniformly distributed random numbers $R_0, \ldots, R_{d-1}$, all taking values from the set $\{0, \ldots, M-1\}$. The i-th comparator outputs a 1 if $R_i < k$ and a 0 otherwise. Thus, the outputs of the d comparators are all random bits with probability $\frac{k}{M}$ of being one. As the counter value increases from 0 to $M-1$, the probabilities of the output bits of all the comparators take value $0, \frac{1}{M}, \ldots, \frac{M-1}{M}$ in sequence, which correspond to all the x points in the numerical integration formula (3). Since the counter is triggered by the pre-counter, the SWU outputs random bits with the same probability $\frac{k}{M}$ for L basic clock cycles. Thus, the total number of basic clock cycles it takes to get the integration result is LM.

The de-randomizer is just an $(l+m)$-bit counter, which counts the number of ones in the output bit stream Y of the SCU. By counting the number of ones, the counter essentially adds together all the values $f(0), f(\frac{1}{M}), \ldots, f(\frac{M-1}{M})$, which are encoded stochastically. The final integration result can be obtained from the final counting value y by dividing y by LM: the division by L transforms the count of ones into a probability and the division by M obtains the final integration result from the sum $\sum_{k=0}^{M-1} f(\frac{k}{M})$. Since $LM = 2^{l+m}$, we can simply obtain the final result $\frac{y}{LM}$ by interpreting y as a binary fraction $\frac{y}{2^{l+m}}$.

In summary, the architecture shown in Figure 4 implements the numerical integration stochastically.

5. EXPERIMENTAL RESULTS

Since the number of random bits L for encoding each value affects the output error of the stochastic implementation, in the experiments, we first study the relation between L and the output error. Further, we analyze the area and the delay of the proposed stochastic implementation and compare these metrics to those of the conventional implementation using binary radix encoding.

5.1 The Relation between L and the Output Error

As we stated in Section 3, the stochastic implementation suffers from error due to randomness, which can be reduced by increasing the number of random bits L for encoding a probability value. As shown in Equation (4), the output error also depends on the number of points M evaluated in the numerical integration. Therefore, in order to determine the value L, we first study how L affects the output error given different choices of M. We obtain this relation using an stochastic architecture with its SCU implementing a Bernstein polynomial of degree 6. To get the statistical results, we randomly choose 100 Bernstein polynomials of degree 6 and with coefficients in the unit interval as the integrand. For each numerical integration instance, we simulate its stochastic implementation 50 times. For each simulation, we obtain the absolute output error as the difference between the value returned by the simulation and the value calculated by Equation (3) using a digital computer. We then average all the output errors over all the simulations and all the polynomials. Table 1 shows the mean absolute output error for each combination of L and M with L taking value from the set $\{1, 2, 4, 8, 16, 32\}$ and M taking value from the set $\{128, 256, 512, 1024, 2048\}$. We also plot the mean absolute error versus different bit lengths L subject to different choices of M in Figure 5. From the figure, we can see that for those combinations of L and M where the products LM are the same, their mean output errors are almost the same. With $LM = 2048$, the mean output error is roughly 1%. If the product LM increases to 8192, the error reduces to 0.5%. Analyzing the data in Table 1, we also find that with the product LM doubled, the mean output error decreases by roughly $\frac{1}{\sqrt{2}}$, which agrees with Equation (4).

Table 1: Mean absolute output error versus the length of the stochastic bit stream L and the number of points M evaluated in the numerical integration.

	M				
L	128	256	512	1024	2048
1	0.0443	0.0312	0.0222	0.0158	0.0111
2	0.0308	0.0222	0.0156	0.0111	0.00759
4	0.0220	0.0155	0.0110	0.00748	0.00509
8	0.0156	0.0109	0.00759	0.00508	0.00352
16	0.0110	0.00761	0.00507	0.00347	0.00232
32	0.00757	0.00502	0.00351	0.00230	0.00115

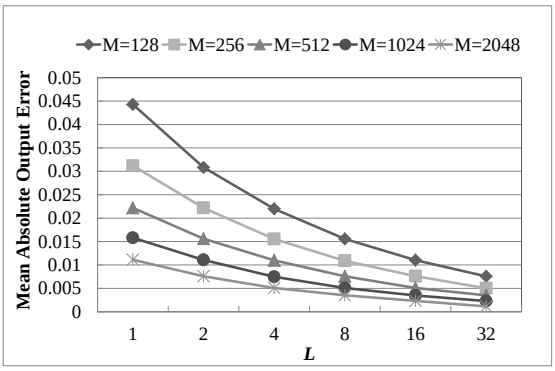

Figure 5: The plot of the mean absolute output error versus the length of the stochastic bit stream L subject to different choices of M.

5.2 The Area and the Delay of the Stochastic Implementation

We estimate the area and the delay of the stochastic implementation of numerical integration in this section. Specifically, we analyze a design with its SCU implementing a Bernstein polynomial of degree 6. Note that the system shown in Figure 4 is built with basic digital modules. We apply the common design for these modules. We estimate the circuit area by counting the number of fanin-2 gates contained in the circuit. These gates include AND, OR, NAND, NOR, XOR, and XNOR gate. We estimate the delay of the circuit by counting the number of fanin-2 gates that lies on the critical path.

The area of each module is listed in Table 2. Note that since the SCU implements a Bernstein polynomial of degree 6, we have 6 comparators in the SWU. The adder has 6 inputs. The multiplexer is a 7-to-1 multiplexer with channel width n. Due to its serial processing, the stochastic implementation also allows us to pipeline it to reduce delay. In our design, we insert two pipelines: the first is inserted between the SWU and the SCU, and the second is inserted between the multiplexer and the comparator in the SCU. The area cost of the pipeline is also listed in Table 2. In our design, we choose the bit width of the inputs C_i as 10, i.e., $n = 10$. Thus, we obtain the entire area of the stochastic implementation as

$$42m + 12l + 336,$$

where $m = \log_2 M$ and $l = \log_2 L$.

Table 2: The areas of the digital modules used in the stochastic implementation shown in Figure 4.

module name	area	comments
pre-counter in SWU	$6l$	$l = \log_2 L$
counter in SWU	$6m$	$m = \log_2 M$
6 comparators in SWU	$6(5m-1)$	$m = \log_2 M$
adder in SCU	17	—
multiplexer in SCU	$18n$	n is the bit width of C
comparator in SCU	$5n-1$	n is the bit width of C
counter in de-randomizer	$6(l+m)$	$l = \log_2 L, m = \log_2 M$
pipeline	$36+6n$	n is the bit width of C

The delay of the comparator in the SWU is $m + 2$. The delay of the combinational logic consisting of the adder and the multiplexer is 9. The delay of the comparator in the SCU is $n + 2 = 12$. Due to the pipeline, the delay of the stochastic implementation is

$$\max\{m+2, 9, 12\} = \max\{m+2, 12\}.$$

Further, since we need $LM = 2^{l+m}$ clock cycles to get the integration result, the total computation time is

$$2^{l+m} \cdot \max\{m+2, 12\}.$$

We want to compare the area cost and the time consumption of our stochastic implementation to those of the conventional implementation using binary radix encoding. We implement the conventional implementation using binary multiplier and adder. We assume that the conventional implementation integrates the same polynomial as the stochastic implementation does. Thus, the integrand is a power-form polynomial of degree 6. A polynomial $f(x) = \sum_{i=0}^{6} a_i x^i$ can be rewritten as

$$f(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_5 + xa_6))),$$

With this rewriting, we can evaluate the polynomial in 6 iterations. This iterative computation only requires one multiplier, one adder, and one register which stores the intermediate value. We build a circuit that combines these three components together. In order to achieve the same precision as the stochastic implementation, the conventional implementation works on binary numbers with $m + l$ bits. The area of the circuit is

$$6(m+l)^2 + 3(m+l) - 6$$

and the delay of the circuit is $4(m+l)-2$. Note that the delay only corresponds to a basic computation of the form $a + b \cdot c$. Therefore, the total time to get the final integration result should be $6M$ times that delay, which is

$$2^m \cdot 6 \cdot (4(m+l) - 2).$$

Therefore, the ratio of the area of the stochastic implementation to that of the conventional implementation is

$$r_a = \frac{42m + 12l + 336}{6(m+l)^2 + 3(m+l) - 6} \tag{6}$$

The ratio of the delay of the stochastic implementation to that of the conventional implementation is

$$r_d = \frac{2^l \cdot \max\{m+2, 12\}}{6(4(m+l) - 2)} \tag{7}$$

Table 3: The area ratio calculated by Equation (6) versus the length of the stochastic bit stream L and the number of points M evaluated in the numerical integration.

	M				
L	128	256	512	1024	2048
1	2.04	1.67	1.41	1.21	1.06
2	1.60	1.35	1.16	1.02	0.906
4	1.29	1.12	0.980	0.872	0.785
8	1.07	0.940	0.839	0.756	0.688
16	0.900	0.805	0.728	0.663	0.609
32	0.772	0.699	0.639	0.587	0.544

We calculate the area ratio r_a and the delay r_d for all the combinations of $L = 2^l$ and $M = 2^m$ with L taking value from the set $\{1, 2, 4, 8, 16, 32\}$ and M taking value from the set $\{128, 256, 512, 1024, 2048\}$. The results are listed in Table 3 and 4.

From Table 3, we can see that the area ratio decreases when either L or M increases. This is because the area of the stochastic implementation increases linearly with either l or m, while the area of the conventional implementation increases quadratically with either l or m. Furthermore, we can see that when $LM \geq 4096$, the area ratio is below 1, which means that the area of the stochastic implementation is smaller than that of the conventional implementation.

From Table 4, we can see that the delay ratio r_d almost doubles with L doubled. This is because the delay of the stochastic implementation increases exponentially with l, while the delay of the conventional implementation increases linearly with l. However, the constant multiplying l in the formula for the delay of the conventional implementation is quite large. Therefore, when $L \leq 16$, the delay ratio is still below 1, which means that the delay of the stochastic implementation is shorter than that of the conventional implementation. Also, we can see that the delay of the stochastic implementation is much shorter than that of the conventional implementation when $L = 1$.

Table 4: The delay ratio calculated by Equation (7) versus the length of the stochastic bit stream L and the number of points M evaluated in the numerical integration.

	M				
L	128	256	512	1024	2048
1	0.0769	0.0667	0.0588	0.0526	0.0516
2	0.133	0.118	0.105	0.0952	0.0942
4	0.235	0.211	0.190	0.174	0.173
8	0.421	0.381	0.348	0.320	0.321
16	0.762	0.696	0.640	0.593	0.598
32	1.39	1.28	1.19	1.10	1.12

Finally, we want to compare the overall performance of the stochastic implementation to that of the conventional implementation. We use the product of the area ratio and the delay ratio as the measure. We list the products for different combinations of L and M in Table 5. From the table, we can see that the products are all less than 1 except when $L = 32$ and $M = 128$, which indicates that the overall performance of the stochastic implementation is better than that of the conventional implementation.

Table 5: The product of the area ratio and the delay ratio versus the length of the stochastic bit stream L and the number of points M evaluated in the numerical integration.

	M				
L	128	256	512	1024	2048
1	0.157	0.111	0.0828	0.0638	0.0547
2	0.213	0.159	0.122	0.0971	0.0854
4	0.304	0.235	0.187	0.152	0.136
8	0.449	0.358	0.292	0.242	0.221
16	0.686	0.560	0.466	0.393	0.364
32	1.07	0.895	0.757	0.648	0.608

6. CONCLUSION AND FUTURE WORK

In this work, we propose a novel implementation of the numerical integration using logical computation on stochastic bit streams. We show through mathematical analysis that by summing a large number of terms in the integration, we can reduce the delay of stochastic implementations significantly. Overall, the stochastic design that we propose in this work has both a smaller area and a shorter delay than conventional implementations based on binary radix encoding. We observe that, similarly, many digital signal processing (DSP) applications are predicated on summing a large number of terms. In future work, we will study how to implement DSP applications through logical computation on stochastic bit streams.

ACKNOWLEDGEMENTS

This work was supported in part by National Science Foundation grant no. CCF-1241987. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

7. REFERENCES

[1] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes: The Art of Scientific Computing (3rd ed.)*. Cambridge University Press, 2007.

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

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

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

[5] B. Brown and H. Card, "Stochastic neural computation II: Soft competitive learning," *IEEE Transactions on Computers*, vol. 50, no. 9, pp. 906–920, 2001.

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

[7] S. Bade and B. Hutchings, "FPGA-based stochastic neural networks — implementation," in *IEEE Workshop on FPGAs for Custom Computing Machines*, 1994, pp. 189–198.

[8] M. van Daalen, P. Jeavons, J. Shawe-Taylor, and D. Cohen, "Device for generating binary sequences for stochastic computing," *Electronics Letters*, vol. 29, no. 1, pp. 80–81, 1993.

[9] V. Gaudet and A. Rapley, "Iterative decoding using stochastic computation," *Electronics Letters*, vol. 39, no. 3, pp. 299–301, 2003.

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

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

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