Parallel Implementation of Finite State Machines for Reducing the Latency of Stochastic Computing

Cong Ma and David J. Lilja
Department of Electrical and Computer Engineering, University of Minnesota, Twin Cities
{maxxx376, lilja}@umn.edu

Abstract—**Stochastic computing, which employs random bit streams for computations, has shown low hardware cost and high fault-tolerance compared to the computations using a conventional binary encoding. Finite state machine (FSM) based stochastic computing elements can compute complex functions, such as the exponentiation and hyperbolic tangent functions, more efficiently than those using combinational logic. However, the FSM, as a sequential logic, cannot be directly implemented in parallel like the combinational logic, so reducing the long latency of the calculation becomes difficult. Applications in the relatively higher frequency domain would require an extremely fast clock rate using FSM. This paper proposes a parallel implementation of the FSM, using an estimator and a dispatcher to directly initialize the FSM to the steady state. Experimental results show that the outputs of four typical functions using the parallel implementation are very close to those of the serial version. The parallel FSM scheme further shows equivalent or better image quality than the serial implementation in two image processing applications Edge Detection and Frame Difference.**

I. Introduction

Stochastic computing has shown to be low cost in terms of hardware area, high fault-tolerance and short critical path compared to computations using conventional binary encoding. Computations based on this stochastic approach can be implemented with very simple logic.

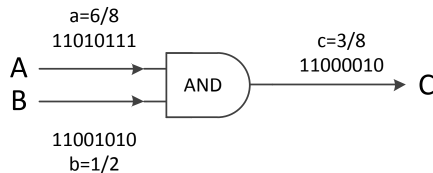


Fig. 1: Stochastic Computing Multiplication using a single AND Gate

Combinational logic has been studied in the early stochastic computing. For instance, an AND gate can be implemented to calculate multiplication as in Fig 1. Stochastic sequential logic using a Finite State Machine (FSM) was first proposed by Brown and Card [1] and then validated by Lilja and Li [2]. The FSM as in Figure 2, consisting of only a few D flip-flops and simple combination logics, is capable of approximating functions, such as exponential, hyperbolic tangent (tanh) and absolute value. However, the stochastic computing will cause long latencies due to its long bit stream [3]. This latency can be reduced by implementing parallel stochastic units when only using the combinational logic [4]. Because any bit in combinational logic without any feedback loop [5] is uncorrelated with each other, we can implement it in serial, which is distributed in time, or in parallel, which is distributed in space. Both will have the same expected output value. On the other hand, the FSM as a sequential logic, having bits correlated in time, cannot be directly implemented in parallel.

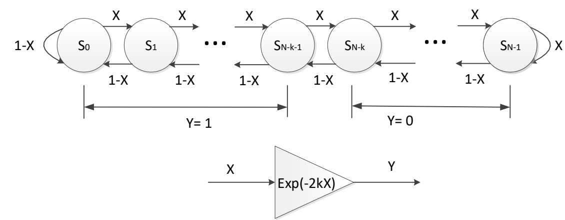


Fig. 2: Finite State Machine diagram for approximating the exp function.

Currently, the length of the stochastic computing bit stream is typically from 256 to 1024 bits, which means the clock frequency should be 256 to 1024 times of the sampling frequency. For example, for audio applications, the sampling rate is around 48kHz, which would require the stochastic computing circuit to boost its clock rate to around 48MHz. This is acceptable for low frequency situations, but for higher frequency applications, this will significantly increase the hardware area and energy use.

In this paper, we focus on how to implement the FSM in parallel to reduce the long calculation latency. We also propose a state dispatcher to quickly put the FSM in steady state, thus making the FSM possible to be implemented in parallel. The rest of the paper is organized as follows. Section II briefly reviews the previous works. A straightforward implementation of parallel FSM is studied and a new structure is proposed in Section III. We present part of the experimental results of the new parallel FSM and the hardware cost comparisons respectively in Sections IV and V, and draw conclusions in Section VI.

II. Related Work

Since the early works of Gaines [6], researchers have employed the stochastic computing algorithms in various

areas including neural networks [7], [8], [3], signal processing [9], [10], [11], [12], [13] and image processing applications [2], [14]. Qian et al. [15] has proposed a synthesis method using the Bernstein polynomials to approximate functions with only combinational logic. However, such synthesis requires multiple uncorrelated random input bit streams, which increases the hardware cost. Besides, to achieve a higher accuracy, the degree of the polynomial will have to increase, which will cause the number of the input sources to grow even larger. Functions such as exponential cannot be efficiently implemented due to the large hardware cost. Brown and Card [1] proposed a sequential logic FSM. Li and Lilja [2] later validated the mathematics of the FSM and proposed systematic methods to synthesize and implement it into various image processing applications [14]. With very limited hardware cost, the FSM is capable of approximating functions such as absolute value, exponential and tanh, and is widely used in various applications [7], [2], [3].

Although these applications benefits from low hardware cost and high fault tolerance of stochastic computing, the long sequence of bits to get a smaller variance for a better estimate of output value creates long latency and significant performance drop compared with conventional implementations. A parallel implementation of combinational logic is proposed in [4] which has higher computing accuracy and faster processing speed by using a nibble serial data organization, but this method can only apply to combinational stochastic logic not sequential ones such as FSM. Wang, et al [5] further studied the impact of feedback loop on stochastic circuits. A rerandomizer is proposed to break the correlation introduced by the feedback loop. Because the rerandomizer generates the bit stream for the next real domain clock value, it must preserve the equivalent precision and uses all the bits from the previous value to generate the next one, which causes a time delay of a real domain clock. Pixel level parallelism that requires a large array of stochastic computing units to calculate the entire image is proposed in [2] to speedup the application processing time. Although this method can improve throughput, the calculation latency for each pixel remains the same.

III. THE PARALLEL FSM DESIGN

The Finite-State Machine is based on the Markov Chain theory. The probability distribution of states after a long run is deterministic and unique for each input value, which is called the steady-state distribution [16]. For example, the transition matrix P of a 4-state FSM with input probability of x is

$$P = \begin{pmatrix} 1-x & x & 0 & 0 \\ 1-x & 0 & x & 0 \\ 0 & 1-x & 0 & x \\ 0 & 0 & 1-x & x \end{pmatrix}$$

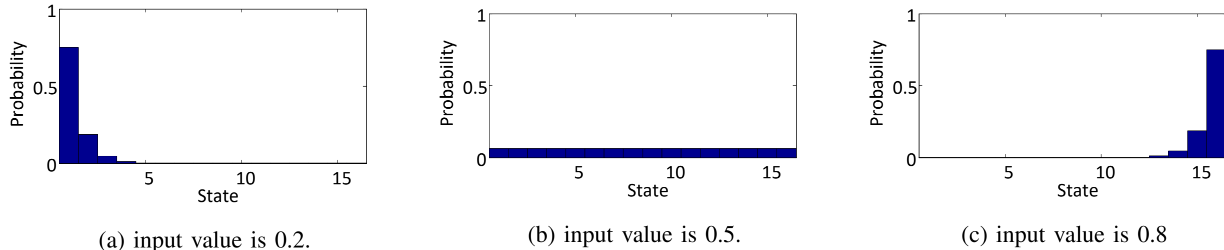
The steady-state distribution π of this transition matrix, as in Equation. 1 can be shown to exist and to be solved.

$$\pi = \pi P \tag{1}$$

Fig. 3 shows the steady-state distributions of a typical 16-state FSM with different input values. The expected time (i.e., number of clock cycles) to reach this steady state can be calculated using the transition matrix. When each vector of P^n becomes the same as the steady-state distribution, n is the number of steps to reach the steady state. A 16-state FSM with an input value of 0.5 can be estimated to require at least 200 cycles to reach the steady state. This will become a huge disadvantage if we want to implement a parallel FSM with bit stream length of 1024, since the convergence period will be too long for each parallel copy. We implemented a straightforward parallel FSM of the absolute value function as in Fig. 6a to show this impact. The inputs are 32 uncorrelated bit streams generated by feeding the same value X into 32 Linear-feedback shift register (LFSR) random bit generators. Then each input is sent to an absolute value function FSM. The mean value of all output bit streams represent the final output value Y. We initiate the FSM with different states 0, 7 and 15, which are the left extreme, middle and right extreme points of all the states. The performance of this straightforward parallel implementation is shown in Fig. 4. When the number of parallel copies is 4 and the length of each bit stream is 256, the output mean value is still close to the real value. However, as the number of parallel copies increases, the output mean value becomes less accurate. When the initial state is 0, the right part of the results significantly drift away from the expected results as parallel copies increases. When the initial is 15, the left part drifts away. When the FSM starts at the middle state 7, all data points move away from the correct value. This is because the FSM generates wrong outputs before it reaches the steady state and this impact grows significantly when the bit stream becomes shorter.

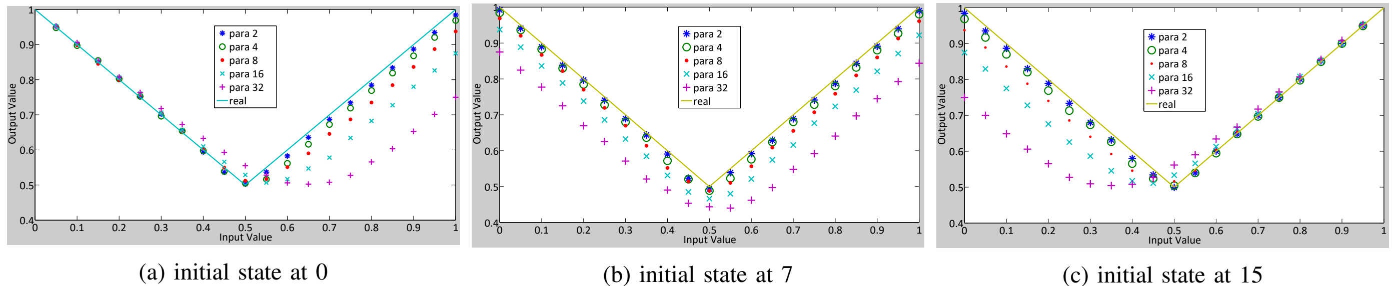
Further, we implemented this straightforward parallel FSM in one of the image applications, Frame Difference, that uses the absolute value function as in [2]. We implemented 32 parallel copies of the FSM units with different initial states at 0, 7 and 15, and each bit stream has 32 bits, so the total number of stream bits of each pixel is 1024, same as the serial stochastic implementation. Fig. 5 shows clearly that the straightforward implementations lose most of the information and fail to compute the correct Frame difference results. Moreover, the results from initial state at 0 and 15 are somehow in complementary shapes. Combining the two can give us a graph very close to the correct result. Whereas when the initial state is at 7, the output shows a rough shape but most of the details are lost. This matches the observations of the absolute value implementation in Fig. 4

This phenomenon is due to the Markov Chain nature of the FSM. Each input value generates a transition matrix that has a steady state distribution as in Fig. 3. When the input value is much smaller than 0.5, the steady state distribution mostly concentrates to state 0. When the input value is much larger than 0.5, it concentrates to state 15. Therefore only half of the outputs are correct when we set the initial state to be 0 or 15 and all outputs are not correct when we set initial state as 7. To faster reach the steady state and decrease the convergence time, we can manually store the steady state distributions and initiate the FSM directly to the them when the input value is



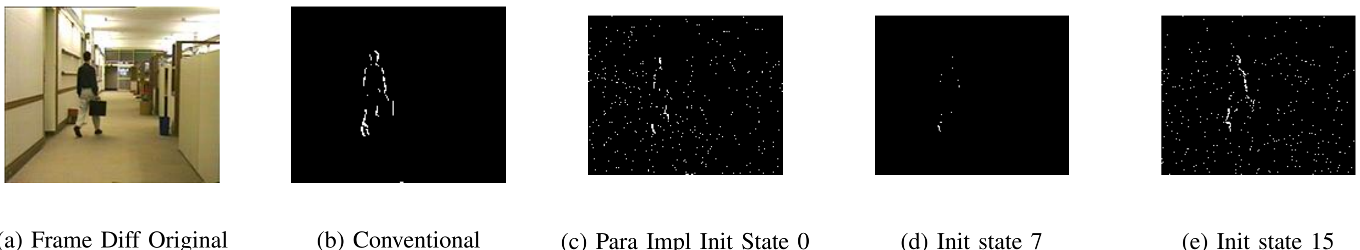
(a) input value is 0.2. (b) input value is 0.5. (c) input value is 0.8

Fig. 3: The Steady-State distribution of a 16-state FSM. From the figure, we can see that this distribution is symmetric about the input value of 0.5. The distribution changes the most here, making it very sensitive around 0.5.



(a) initial state at 0 (b) initial state at 7 (c) initial state at 15

Fig. 4: The mean output of the straightforward implementation of FSM with 2, 4, 8, 16 and 32 parallel copies. The FSM is a typical 16-state absolute value function. The three subgraphs are using different initial state at 0, 7 and 15 respectively.



(a) Frame Diff Original (b) Conventional (c) Para Impl Init State 0 (d) Init state 7 (e) Init state 15

Fig. 5: Simulation results of the conventional deterministic scheme, serial and a straightforward parallel stochastic implementation on Frame Difference with different initial states. The straightforward parallel stochastic implementation is clearly not able to compute the correct results.

known. A dispatcher is proposed to initiate the FSMs from any input value as in Fig. 6d.

The dispatcher itself is a look up table (LUT), through which the input value can pick its corresponding set of initial states. For example, when the input is 0.5, the initial states are evenly distributed among all FSMs as in Fig. 3b. Each state (of 16 states) will be assigned to two FSMs as initial state for a 16-state parallel FSM of 32 parallel copies to mimic the steady distribution of 0.5. Thirty-two initial states are stored in the table to approximate the distribution from each input. The LUT has 20 entries from 0 to 1, with step of 0.05. The dispatcher requires an estimation of the input value to pick the correct entry of states, so an estimator is implemented before the dispatcher unit.

We propose two implementations of the estimation unit, parallel counter and majority gate counter, as in Fig.6c. Because the dispatcher LUT entries increase with a step of 0.05, we choose the estimation bit stream length to be 416 (32 parallel copies x 13 cycles) where the standard deviation of the estimation is 0.025 ($est \pm 0.025$) to be precise enough for the parallel counter to pick the entries in the dispatcher. Moreover, since the steady state distribution is less sensitive around input 0 and 1, a simpler majority gate as in Fig.7 can meet the estimation needs as well. During the estimation process, the dispatcher could not give an initial-states set, the parallel FSM unit will have a stall. We could store the bits in the estimation process and feed them back to the FSMs in the next estimation, making it a simple pipeline to avoid this stall. This way, it only impacts the first input data and does not slow down the overall calculation speed. The complete parallel FSM implementation is shown in Fig.6b.

IV. Experiments and Results

In this section, we will introduce the experimental methodology and present the computational results. We first set up the parallel FSM unit to approximate 3 typical FSM functions, absolute value, tanh and exponential to study the parallel implementation impact. We implemented our scheme with 32

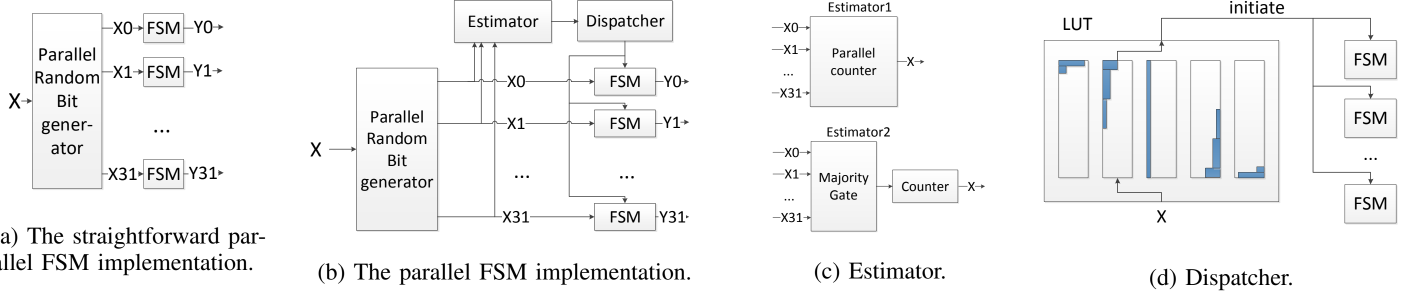


Fig. 6: The straightforward parallel FSM implementation and the proposed parallel implementation. The proposed parallel FSM has 32 parallel short bit streams sent to the Estimator to obtain an initial guess for the input. Two Estimator implementations are parallel counter and majority gate counter. This initial estimate is then sent to the Dispatcher to look up a set of state configurations to initialize the parallel FSMs.

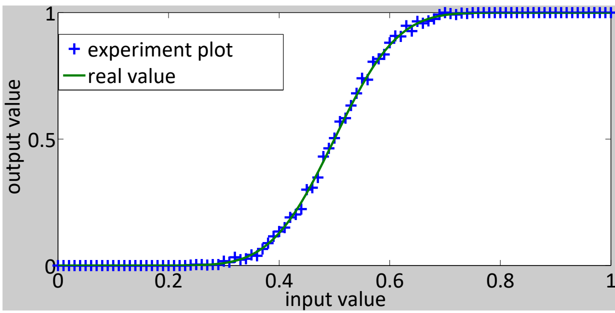


Fig. 7: The experiment and analytical output result of a 32-input Majority Gate.

parallel units of FSMs that can reduce the bit stream length from 1024 to 32. Due to the probabilistic nature of the stochastic computing scheme, we perform our experiments repeatedly for 10 times for statistical significance. The experimental results compare the accuracy and consistency among different schemes, including the serial FSM, the straightforward parallel FSM as in Fig. 6a, the parallel FSM with a parallel counter estimator and a majority gate estimator as in Fig. 6c. Then, we implemented our parallel FSM with the majority gate estimator into two image processing applications as in [2]. Edge detection uses a FSM to approximate absolute value function. And Frame Difference applications uses two FSMs to approximate absolute value and tanh functions respectively. We implemented the FSMs in parallel the same way as in the previous single function implementations. The experimental results compare the output image quality using mean squared error (MSE) and peak-signal-to-noise ratio (PSNR) among the conventional deterministic scheme (Conventional), the serial stochastic scheme (Serial) and parallel stochastic scheme (Parallel). The MSE is the mean of the square of each pixel error between the stochastic implementation and the conventional implementation results as in Equation 2.

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(I(i,j) - K(i,j))^2 \qquad (2)$$

where I refers to the image result from the conventional implementation and K refers to either of the stochastic imple-

TABLE I: The average error and deviation of the parallel FSMs.

	Abs err	Abs std	Tanh err	Tanh std	Exp err	Exp std
serial	0.0066	0.0127	0.0121	0.0224	0.0105	0.0171
straight	0.1320	0.0118	0.0840	0.0234	0.1500	0.0141
paraCnt	0.0038	0.0141	0.0049	0.0248	0.0176	0.0211
mjrEst	0.0057	0.0150	0.0045	0.0285	0.0199	0.0238

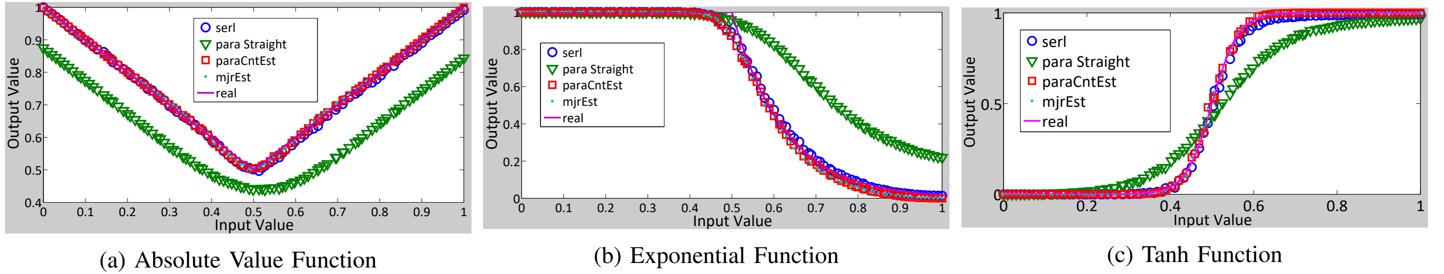
mentations. PSNR is further calculated using the MSE as in Equation 3.

$$PSNR = 20 \times log_{10}(MAX_I) - 10 \times log_{10}(MSE) \qquad (3)$$

where MAX_I is the maximum possible pixel value of the image. The bit length of each pixel of all images in this paper is 8, so the MAX_I value is $2^8 - 1 = 255$. PSNR shows the ratio between the maximum possible power of a signal and the noise in dB.

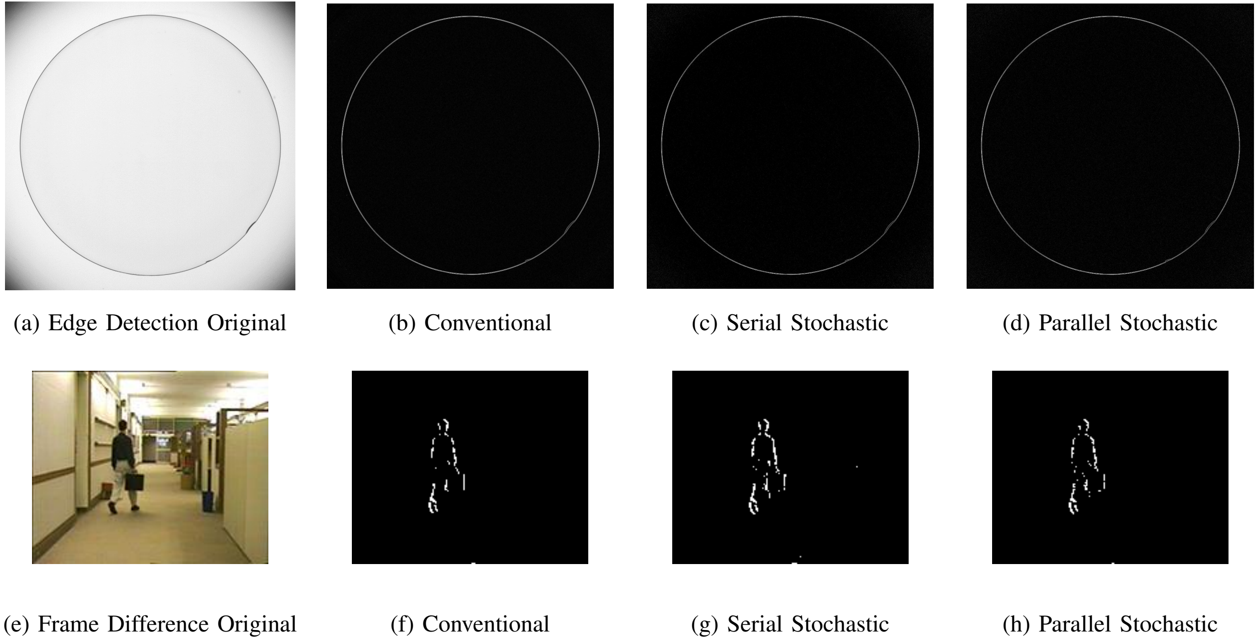
Fig. 8 compares the output accuracy, showing the true output value and the mean value of the repeated experimental results of the serial and different parallel stochastic FSM implementations. The average error and the standard deviation of each implementation are shown in Table. I. The straightforward scheme shows significant difference from the true output, while the other two parallel schemes with estimator and dispatcher are very close to the true output, showing very good accuracy. The error becomes bigger as the input value grows to 0.5 due to larger autocorrelation and variance impacts [17]. The parallel FSM tend to be closer to true value than the serial FSM when the input value is near 1, especially with the exponential and tanh functions. Since the serial FSM initiates to state 0, it needs time to grow from state 0 to 15 when the input value is close to 1. This transition requires at least 16 steps, generating 16 wrong output bits. A bit stream of 1024 bits has an error rate of $\frac{16}{1024} = 1.56\%$ with 16 wrong output bits. However, the parallel FSM does not fix the initial states, making no difference between different input values. This makes the parallel implementation more accurate near input value of 1.

Edge Detection and Frame Difference results are shown in Figures. 9. It is hard to visually find any difference in both



(a) Absolute Value Function (b) Exponential Function (c) Tanh Function

Fig. 8: The output mean value of two parallel FSMs with parallel counter as estimator or majority gate estimator and the serial FSM. Both estimators use 13 clocks, $13 \times 32 = 416$ bits, to approximate the input value.



(a) Edge Detection Original (b) Conventional (c) Serial Stochastic (d) Parallel Stochastic

(e) Frame Difference Original (f) Conventional (g) Serial Stochastic (h) Parallel Stochastic

Fig. 9: Simulation results of the conventional deterministic scheme, serial and parallel stochastic implementation on Edge Detection and Frame Difference. The three schemes on both applications show almost no visual differences.

TABLE II: The MSE and PSNR of image processing applications

Applications	MSE		PSNR (dB)	
	serial	parallel	serial	parallel
EdgeDetect	47.3	47.9	31.4	31.3
FrameDiff	156.5	133.4	26.2	26.9

application results other than some outliers due to the probabilistic nature of the stochastic computing. Detailed image quality comparisons are listed in Table II. Both applications achieved acceptable PSNR of 8-bit images under both serial and parallel implementations [18]. Both of them have similar MSE under two different schemes.

In summary, the experimental results show that the performance of the parallel FSM is as good as the serial implementation. The simplified majority gate estimator can also compete with a more complex parallel counter, which suggests more simplifications could be exploited for low accuracy applications. When number of parallel units increases and the number of each bit stream decreases, this estimator-dispatcher mechanism becomes crucial to ensure the accuracy of the parallel FSM scheme. The image processing applications show that the parallel FSM implementation can achieve equivalent or better image quality than serial implementations.

V. Latency and Hardware Cost

We implemented the serial and parallel stochastic finite-state machine in verilog using Xilinx ISE. The estimator of the parallel FSM unit is implemented using two different schemes, parallel counter and majority gate as in Figure 6c. The hardware cost of the 32-degree parallelism implementation is shown in Table III, where parallel (PC) refers to parallel FSM implementation using parallel counter as the estimator and parallel FSM (MG) refers to the implementation using majority gate. The hardware area reported is the number of look-up table and flip-flop pairs (LUT-FF).

Although the parallel implementation of the FSM introduces hardware overhead, it reduces the latency compared to the serial version. For instance, when the number of parallel copies is 32, the serial implementation latency reduces from 1024 cycles to 32 cycles. This significant latency reduction can be critical for high frequency applications. Although the parallel

TABLE III: Hardware Cost, Latency and Area-Delay Product of serial and parallel FSM with 32 degrees of parallelism

	serial	parallel (PC)	parallel (MG)
FSM unit	4	8×32	8×32
supporting unit	-	72	66
total LUT-FF pairs	4	328	322
initial latency	0	13	13
latency	1024	32	32
Area-Delay Product	4096	10496	10304

implementation does introduce an initial latency during the input estimation process by 13 cycles as in Table III, we can minimize this impact by feeding back these 13 bits during the next estimation as a pipeline. We can further see that the parallel implementation Area-Delay Product (ADP) is greater than the serial ADP due to the significant hardware overhead for the parallel implementation. Of course it is a common practice to trade off area for better performance. Each FSM unit of the parallel implementation becomes about twice as large as the serial FSM, which contributes the larger hardware overhead. This is because the parallel FSM must be able to initialize to different states, which increases the hardware complexity and area cost. Another hardware overhead of the parallel FSM comes from the dispatcher and estimator, which is shown in Table III as supporting unit. The dispatcher is simply a look-up table (LUT) with multiple entries, that each store a set of initial states for the number of parallel FSM units. As for the estimator implementation, we can further see that the majority gate scheme reduces the supporting unit area significantly compared to the parallel counter scheme in the table.

VI. CONCLUSION

This paper proposed a parallelization scheme for the stochastic computing sequential logic, the Finite State Machine. Using a look-up table dispatcher to set the initial states of multiple FSMs, the parallel FSM can immediately work from the steady state, avoiding the long convergence period. We also proposed two kind of estimators for the dispatcher. One of them, the parallel counter, requires larger hardware area, but provides better estimation of the input value. The other, using the majority gate, naturally fits the trend of steady-state distribution with the input values, and also simplifies the estimator hardware.

The proposed parallel scheme can effectively implement the stochastic sequential logic FSM in parallel to reduce the long calculation latency with some hardware overhead. Experiments on three typical FSM functions show that the accuracy and variance of the parallel FSM scheme are comparable to the serial implementation. The parallel FSM scheme further shows equivalent or better image quality than the serial implementation in two image processing applications. We conclude that quickly initializing the FSM by estimating the initial state using only a few bits of the input value allows parallelism to be effectively exploited in stochastic logic that uses storage elements.

VII. ACKNOWLEDGEMENT

We thank the anonymous reviewers for their comments towards improving this paper. We are grateful for resources from the University of Minnesota Supercomputing Institute. 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.

REFERENCES

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

[2] P. Li and D. Lilja, “Using stochastic computing to implement digital image processing algorithms,” in *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pp. 154–161, IEEE, 2011.

[3] Y. Ji, F. Ran, C. Ma, and D. Lilja, “A hardware implementation of a radial basis function neural network using stochastic logic,” in *Design, Automation Test in Europe Conference Exhibition (DATE), 2015*, pp. 880–883, March 2015.

[4] L. Miao and C. Chakrabarti, “A parallel stochastic computing system with improved accuracy,” in *SiPS 2013 Proceedings*, pp. 195–200, Oct 2013.

[5] Z. Wang, N. Saraf, K. Bazargan, and A. Scheel, “Randomness meets feedback: Stochastic implementation of logistic map dynamical system,” in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–7, June 2015.

[6] B. Gaines, “Techniques of identification with the stochastic computer,” in *Proc. IFAC Symp. Problems of Identification*, pp. 1–18, 1967.

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

[8] H. Li, D. Zhang, and S. Y. Foo, “A stochastic digital implementation of a neural network controller for small wind turbine systems,” *IEEE Transactions on Power Electronics*, vol. 21, pp. 1502–1507, September 2006.

[9] J. G. Ortega, C. L. Janer, J. M. Quero, J. Pinilla, and J. Serrano, “Analog to digital and digital to analog conversion based on stochastic logic,” in *IEEE 21st Annual Conference of Industrial Electronics Society, IECON’95*, pp. 995–999, 1995.

[10] C. L. Janer, J. M. Quero, J. G. Ortega, and L. G. Franquelo, “Fully parallel stochastic computation architecture,” *IEEE Transactions on Signal Processing*, vol. 44, pp. 2110–2117, August 1996.

[11] S. S. Tehrani, S. Mannor, and W. J. Gross, “Fully parallel stochastic ldpc decoders,” *IEEE Transactions on signal processing*, p. 11, November 2008.

[12] M. Hori and M. Ueda, “Fpga implementation of a blind source separation system based on stochastic computing,” in *IEEE Conference on Soft Computing in Industrial Applications, SMCia’08*, pp. 182–187, 2008.

[13] N. Saraf, K. Bazargan, D. Lilja, and M. Riedel, “Iir filters using stochastic arithmetic,” in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pp. 1–6, March 2014.

[14] P. Li and D. Lilja, “A low power fault-tolerance architecture for the kernel density estimation based image segmentation algorithm,” in *Application-Specific Systems, Architectures and Processors (ASAP), 2011 IEEE International Conference on*, pp. 161–168, IEEE, 2011.

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

[16] A. A. Markov, “Extension of the limit theorems of probability theory to a sum of variables connected in a chain,” *reprinted in Appendix B of: R. Howard. Dynamic Probabilistic Systems, volume 1: Markov Chains. John Wiley and Sons*, 1971.

[17] C. Ma, P. Li, and D. J. Lilja, “Autocorrelation study for finite-state machine-based stochastic computing elements,” in *Proceedings of the International Worshop on Logic Synthesis (IWLS)*, 2013.

[18] N. Thomos, N. V. Boulgouris, and M. G. Strintzis, “Optimized transmission of JPEG2000 streams over wireless channels.,” *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, vol. 15, pp. 54–67, Jan. 2006.