i-ETC: ISEL Academic Journal of Electronics, Telecommunications and Computers
IoT-2018 Issue, Vol. 4, n. 1 (2018) ID-8

Stochastic Theater: Stochastic Datapath Generation Framework for Fault-Tolerant IoT Sensors

Rui Policarpo Duarte[a], Mário Véstias[b], Carlos Carvalho[c], João Casaleiro[c]
[a]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
[b]INESC-ID, Instituto Superior de Engenharia de Lisboa, Portugal
[c]CEDET, Instituto Superior de Engenharia de Lisboa, Universidade Nova, Portugal
rpd@inesc-id.pt mvestias@deetc.isel.pt cfc@cedet.isel.ipl.pt joao.casaleiro@cedet.isel.ipl.pt

***Abstract*— Stochastic Computing has emerged as a competitive computing paradigm that produces fast and simple implementations of arithmetic operations, while offering high levels of parallelism, and graceful degradation of the results when in the presence of errors. IoT devices are often operate under limited power and area constraints and subjected to harsh environments, for which, traditional computing paradigms struggle to provide high availability and fault-tolerance. Stochastic Computing is based on the computation of pseudo-random sequences of bits, hence requiring only a single bit per signal, rather than a data-bus. Notwithstanding, we haven't witnessed its inclusion in custom computing systems. In this direction, this work presents Stochastic Theater, a framework to specify, simulate, and test Stochastic Datapaths to perform computations using stochastic bitstreams targeting IoT systems. In virtue of the granularity of the bitstreams, the bit-level specification of circuits, high-performance characteristics and reconfigurable capabilities, FPGAs were adopted to implement and test such systems. The proposed framework creates Stochastic Machines from a set of user defined arithmetic expressions, and then tests them with the corresponding input values and specific fault injection patterns. Besides the support to create autonomous Stochastic Computing systems, the presented framework also provides generation of stochastic units, being able to produce estimates on performance, resources and power. A demonstration is presented targeting KLT, typical method for data compression in IoT applications.**

Keywords: IoT, FPGA, Fault-Tolerant Computing, Stochastic Bitstreams, Approximate Computing.

I. Introduction

Data intensive Digital Signal Processing (DSP) applications for near real-time image and video processing, neuromorphic [21] and bio-inspired systems [17], are characterized for their regularity in their datapath. Their computations are mainly based on multiplications followed by accumulations, and by the fact that they can tolerate some errors in their computations.

To alleviate Edge servers from the work of computing basic, but essential, DSP and Machine Learning (ML) functions, there is interest in delegating such computing to the Internet of Things (IoT) device. However, in the IoT context, devices are often required to operate under heavy power and area constraints and subjected to harsh environments, struggle to provide high availability and fault-tolerance. To overcome such limitations, this work proposes to make use of a different computing paradigm that blends well with the IoT context, and offers direct analog sensor interface without Analog-to-Digital Converters (ADCs), fault-tolerance and savings in resources and power.

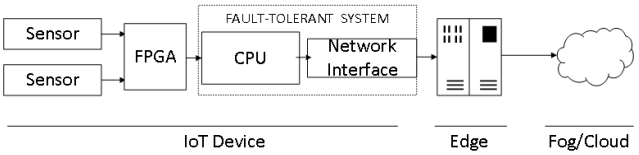

Fig. 1. Illustration of two scenarios for a typical IoT application, with stochastic computing.

Stochastic arithmetic has emerged as an alternative computational paradigm able to provide approximate computations requiring less hardware, towards a circuit design with simpler but massively parallel components, trading off precision for computation time [11].

Applications like neural networks [24], [23], high-throughput Bayesian inference [15], image and video processing [19], Finite Impulse Response (FIR) [5] and Infinite Impulse Response (IIR) [20] digital filters, and autonomous cyber-physical systems [10] are characterized for their regularity in their datapath. Their computations are mainly based on multiple multiplications followed by accumulations. Moreover, many of these applications do not require exact results and can tolerate some deviations in their computations.

The operation of Stochastic Computing (SC) is suitable for reconfigurable devices such as Field-Programmable Gate Arrays (FPGAs) given that the bit level specification of stochastic bitstreams makes it favorable for implementation on these devices.

Figure 1 illustrates the scenario where a dependable IoT system requires the implementation of fault-tolerance mechanisms at all levels of the system, even though it only acquires data from the sensor and communicates the data to the Edge servers. The proposed approach, using SC, performs computations directly over the acquired stochastic bitstream, thus alleviating the computational load at the Edge, and reducing the require fault-tolerance mechanisms at the IoT and Edge levels.

However, the majority of research conducted on SC is confined to a set of applications, which are highly customized, specific to certain applications and difficult to extend its adoption. Furthermore, the benefits of SC are not always clear due to the resources of the supporting elements and the clock latency to process long bitstreams. Often, the benefit of SC is

shadowed by the latency and resources required to interface a traditional computing systems.

As an inspirational example, the Bayesian inference system presented in [7] requires 597 Logic Elements (LEs) to be implemented, of which, only 42 are spent on the datapath for the Bayesian Machine. The remaining 555 LEs are spent on conversion of 13 stochastic bitstreams. The contribution in [18] presents a comparison of parallel binary versus stochastic implementations for neural networks on reconfigurable hardware. The authors concluded that even though stochastic bitstreams require more clock cycles to compute than binary, the advantage of compact realizations in hardware surpasses that through comparison of geometric mean of the two metrics. Therefore there is a need for a methodology to make this assessment at an earlier stage of the design process.

The main claim addressed in this paper is to ease the definition and evaluation of a Stochastic Datapath (SD) to compute, at the IoT-level, mathematical expressions as alternative to other time consuming and prone-to-error design approaches, and without having to delve into the technicalities of High-Level Synthesis (HLS).

This work presents Stochastic Theater, a highly customizable and scalable framework that given a problem's specification as mathematical expression, it generates the corresponding SD, and its supporting blocks, targeting reconfigurable logic. This work is intended to facilitate automated architectural changes via unified and regular interfaces, and design-space exploration often sought in research due to the long execution times. Moreover, this work provides an estimate of resources, power and performance metrics. This enables the usage of the SC in stand-alone stochastic systems or accelerators for heterogeneous and System-on-a-Chip (SoC) platforms.

Stochastic Theater is a novel framework for prototyping of SC systems on FPGAs, and it is an improvement over previous work found in [8]. Moreover, the Karhunen-Loève Transform (KLT) algorithm can be implemented as inner product, which is the same for a FIR filter, hence demonstrating its wide range of applicability of the proposed work.

Stochastic Theater offers the following features:

- scalable and fully automated process through the execution of configuration scripts, to enhance the use of Stochastic Computing;
- generation of custom Stochastic Computing elements, e.g. arithmetic units with more than 2 inputs;
- support for fault injection, through simulation, to predict the behaviour of the stochastic system when operating under fault conditions;
- supports for Self-Timed Ring-Oscillator (STRO) to minimize temporal correlations of bitstreams, according to [9];
- support for different FPGA device families. Currently Cyclone III, IV and V from Altera are supported, but can be easily ported to devices from other vendors.

The flow of Stochastic Theater is illustrated in Figure 2. It begins with a high-level specification from the user as a mathematical expressions described in Python, which are translated into a computational stack, as sequences of inter-connected operands and operators. The framework then generates the Register Transfer Level (RTL) in Very High Speed Integrated Circuits (VHSIC) Hardware Description Language (VHDL) for custom SC arithmetic units, the SD which implements the desired functionality, and the supporting blocks.

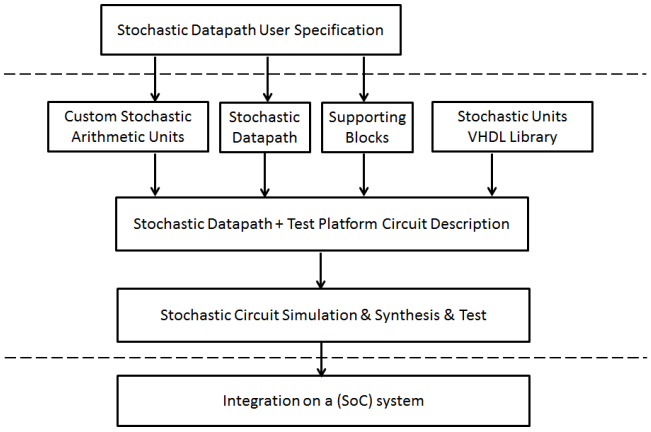

Fig. 2. Flow of the proposed framework to simulate, synthesize and evaluate Stochastic Computing systems on FPGAs.

This paper is organized as follows: section II is devoted to introduce SC and presents the most relevant research contributions incorporated in the proposed framework. Section III presents the details about the proposed framework along with the inner workings of the proposed framework, to generate VHDL entities and the datapath for the mathematical expression to be implemented. A demonstration of a stochastic system with the first implementation of the KLT implemented on an FPGA is in Section IV. Analysis on the outcomes are in section V Conclusions and final remarks are in section VI.

II. Background

J. Von Neumann introduced SC in [22] as a method to design probabilistic logic circuits and synthesize robust systems from unreliable components. In [12], Gaines has introduced the use of stochastic bitstreams to represent operators with high levels of error tolerance.

A. Stochastic Bitstreams

By definition, a stochastic signal is the result of a continuous-time stochastic process which produces two values: 0 and 1. According to [11], a unipolar stochastic bitstream is a sequence of stochastic signals over time whose value is within $[0;1]$ and defined as the number of ones (o) over the total number of bits (t). In bipolar representation, the value is within $[-1;1]$ and is also encoded as a ratio but followed by a negative bias and a scale factor of 2. On stochastic bitstreams there are no weights in the representation, as in typical binary-radix representation, thus all bits have the same contribution for the encoded value. For example, the same sequence of 8 bits 01110110 represents $5/8 = 0.625$ in unipolar and $2 * (5/8 - 0.5) = 0.25$ in bipolar. Figure 3 illustrates the aforementioned stochastic bitstream. On top, there is the clock signal, to ensure synchronism; and on bottom the encoded value.

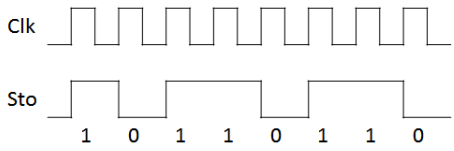

Fig. 3. Example of a stochastic bitstream encoding 0.625 and 0.25 in unipolar and bipolar encodings, respectively.

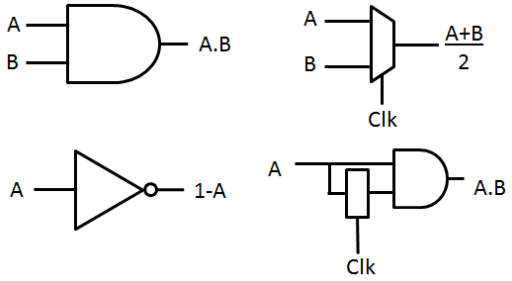

Fig. 4. Block diagram of the unipolar stochastic units: a) multiplier (top-left), b) adder (top-right), c) negation (bottom-left) and d) squarer (bottom-right).

B. Stochastic Arithmetic

Stochastic arithmetic supports basic arithmetic computations like addition and multiplication, as illustrated in Figure 4. Details on stochastic arithmetic units can be found in the survey presented in [2] which covers the most common arithmetic units.

The unipolar stochastic multiplication is only the result of a logic AND of its stochastic inputs. The complement is the negation of the bitstream. Bipolar multiplication is achieved through an XNOR operation. Addition, or more precisely average, is obtained via a round-robin multiplexation of the stochastic inputs, which depends on a N-module counter corresponding to N inputs in the multiplexer. The square of a stochastic bitstream is the equivalent to a multiplication of a bitstream by itself, delayed by a clock cycle. The clock cycle makes the pseudo-random bitstreams to be uncorrelated. The multiplication is now of two independent streams but with the same value, resembling the power of two operation.

The implementation of n-ary add or multiply operators is achieved by adding additional inputs to the logic circuits. In terms of FPGA implementation it means that the number of inputs of a LE can be evaluated simultaneously.

For more complex operators, such as exp, $tanh$ and abs, there are realizations of stochastic operators using Finite State Machines (FSMs). Implementations of such units can be found in [4], [14].

Table I illustrates the sensitivity to temporal correlations, for the case of multiplications of two bitstreams. In this case both input streams encode the value 0.5, which is represented by the same number of 0s and 1s on the bitstream. However, because of the alignment of the bits, the multiplication, which is achieve at the logic AND, produces a bitstream with only 0s, encoding value 0, rather than the expected value of 0,25.

To improve the statistical quality of the stochastic bitstreams on the datapath, this work adopts the STRO proposed in [7].

Bit Num	8	7	6	5	4	3	2	1
A	0	0	1	1	0	0	1	1
B	1	1	0	0	1	1	0	0
A.B	0	0	0	0	0	0	0	0

TABLE I
EXAMPLE OF A MULTIPLICATION OF TWO CORRELATED BITSTREAMS, PRODUCING A BAD RESULT.

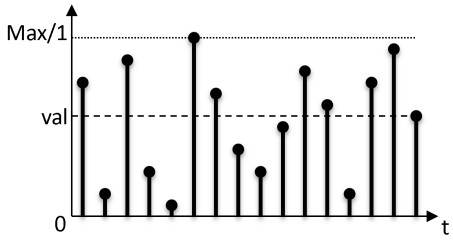

Fig. 5. Detail on process of generating a pseudo-random bitstream for a given binary-radix value between 0 and 1.

The main reasons to consider a STRO instead of a global clock source are: different clock signal for each stochastic unit; all generated clock signals with the variation of voltage, temperature, location on the device and its degradation. All synchronous stochastic units have an instance of this unit to generate its clock signal.

C. Interface and Other Supporting Blocks

Forasmuch as most systems usually use parallel binary-radix representation, it is therefore required a converter from-to stochastic bitstreams to ensure inter-operability. The process of generating the stochastic bitstream is illustrated in Figure 5, where a specific binary-radix value (*val*) is compared with the output of a uniform pseudo-random generator, usually a Linear Feedback Shift-Register (LFSR) [3]. Whenever the pseudo-random number is smaller, it produces a 1 and 0 otherwise. After each pseudo-random sample the ratio between the number of ones and the total number of bits will be towards *val*. In this example, the encoded value is $9/16 = 0,5625$. The conversion from stochastic-to-binary is based on the integration of the 1s on a bitstream, which is accomplished using a binary-radix counter. A second counter is required to count the total number of bits. Figures 6 and 7 show the details of the conversion units.

D. Fault-Tolerance

The graceful degradation of stochastic bitstreams is referred to the impact of bit-flips on the bitstream. In such occurrences,

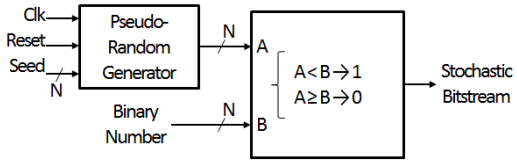

Fig. 6. Block diagram of a binary-to-stochastic unit.

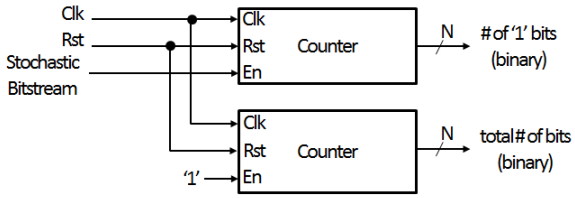

Fig. 7. Block diagram of a stochastic-to-binary unit.

and regardless of the position of the bit on the bitstream, the value of the error associated with each bit-flip is the same as the least significant bit, in binary-radix. On this account, [19] has applied the concept of stochastic logic to a reconfigurable architecture that implements image processing operations on a simulated datapath. The authors show that the quality of the results degrades gracefully with the increase of errors on the bitstream.

E. Disadvantages

The main disadvantages of SC are: a linear increase in the precision of typical binary representations, for stochastic computations it imposes an exponential increase in the length of the bitstream; sensitivity to temporal correlations; and the supporting blocks are usually the performance bottleneck, rather than the arithmetic units.

III. Stochastic Theater: Framework for Specification of Stochastic Datapaths

This work proposes a method to specify it as a mathematical expression, defined as a list of operands and operators, organized in a stack to resemble Reverse Polish Notation (RPN), or postfix notation. The advantages of such representation are: simplified representation without parenthesis, hence fewer operations are needed, faster introduction by the user and with fewer mistakes [13], [1]. In RPN the operators follow the operands. The strength of this notation is the support of n-ary operators, which is compatible with the aforementioned stochastic operators. Example: the computation of $1 - 2 \times 3$ is defined in RPN as $1\ 2\ 3 \times -$. Essentially, the framework recognizes the different operands and operators of a mathematical expression, and then generates the corresponding VHDL. This regular form is easily mapped into an FPGA, exploiting the parallelism offered. From this data structure it is possible to identify the requirements for a system, namely: the number of input, internal and output signals; the different types, number of input operands and data dependencies of the operators used. The data structure is organized as a tree of computations which maintains the data dependencies in the datapath. These mathematical expressions can be variable in size and type of operations.

Considering the following example of a function to be implemented to compute data from 4 sensors:

$$func = \frac{1}{N}\left(i_0 \times i_1 + i_2 \times i_3 \times i_4\right) \tag{1}$$

it has the corresponding RPN stack representation:

$$func = i_0\ i_1\ \times\ i_2\ i_3\ i_4\ \times\ \times\ +\ N\ / \tag{2}$$

To ease the stack manipulation, it is split into a set of partial computations stored in intermediate variables:

$$aux_0 = i_0 \times i_1 \tag{3}$$

$$aux_1 = i_2 \times i_3 \times i_4 \tag{4}$$

which the original expression can be replaced with:

$$func = \frac{1}{N}\left(aux_0 + aux_1\right) \tag{5}$$

and to facilitate the generation of the VHDL source file describing the datapath to implement this expression, it is expressed as a list of operands and operators in Python, e.g. sums and multiplications, resembling RPN. This regular form is easily extracted and can be efficiently mapped into an RTL specification, exploiting the parallelism offered by FPGAs.

The user input for equation 2 in Python can be described by the following list of computations, which itself can be comprised of other lists, or operands, and operators:

```
aux1 = ['i0', 'i1', '*'];
aux2 = ['i2', 'i3', 'i4', '*'];
f = [aux1, aux2, '+'];
```

which results in the following Python variable:

```
>>> f
[['i0', 'i1', '*'], ['i2', 'i3', 'i4',
'*'], '+']
```

Variables `t1` and `t2` are lists of strings, which represent partial computations. These variables can be of any size. The last element, or tail, of the list holds the representation of the operation. In this example, the operands are: `+` or `*`. The remaining elements are the operands. It is also possible to define operations which depend on the results of previous computations, e.g. `f` is defined as the sum, or average, of `t1`, and `t2` . Table II lists the stochastic combinatorial and sequential operators supported so far.

TABLE II
List of the stochastic operators supported.

Operator	Codification
Sum (average)	+
Multiplication	*
Negation	–
Square	pow2
Complement	not

The inputs and outputs of the SD correspond to the number of variables and are determined by the framework. To complete the specification of a datapath it is necessary to indicate the length and type (unipolar/bipolar) of the bitstream. To serve this purpose there is a variable in Python which holds this configuration. However, the architecture of the SD is independent of the bitstream's length.

One of the key strengths of the proposed framework is that given any mathematical expression, regardless the complexity of the mathematical expressions, the system maintains its regularity. The framework integrates the translation of the

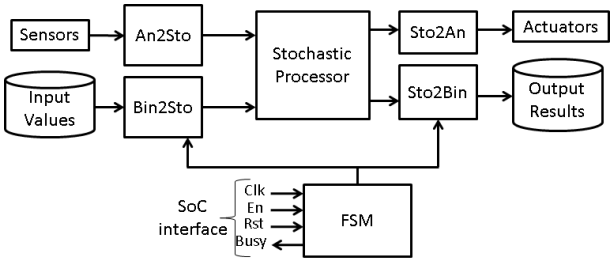

Fig. 8. Top-level architecture of the circuit design to test the Stochastic Datapaths, including the supporting units.

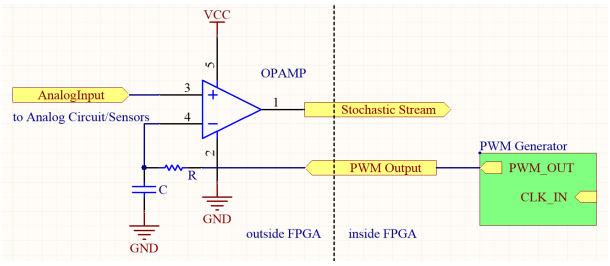

Fig. 9. Analog-to-Stochastic bitstream conversion circuit (from [10]).

expression of the SD into a stack. Apart from the core of the SD, which is different for all expressions, all other supporting units have the same architecture, such as data sources and sinks for the stochastic bitstreams, varying only the number of bits, or the length of the bitstreams supported.

The generated SD was planned to be autonomous or part of a larger system, as illustrated in Figure 8. The SD is in the middle and the rest of the circuit is formed by the supporting units to do the computations. The system is interfaced via the input and output bitstreams, and also the FSM's control signals, namely *Clk*, *Enable* and *Reset*. In particular, the FSM is responsible for the generation of the control signals for all units in the design. It also controls the burn-in period to compensate the clock cycles required by the FSM-based stochastic arithmetic units.

A. Stochastic Arithmetic Units

Typical parallel binary-radix representation all basic operators are either unary or binary, with 1 or 2 operands, respectively. However, has n-ary SC operators support for more than two operands. Therefore, it is required to create the customized components, as it is difficult to account for all possible operators with any number of operands in advance. Therefore, the framework determines the number of arguments for multiplications and sums and then generates the required stochastic arithmetic components. In more detail, it iterates over the aforementioned list of computations to retrieve the different operators and then generates the VHDL entity matching the operation and the number of inputs.

In SC, each operator can have a diverse number of operands. Therefore, it is necessary to generate custom arithmetic units according to the mathematical expression. Moreover, the number of variables considered is unknown, so it is also necessary to create the interfaces to support any number of inputs and outputs. The VHDL source files are created, and used to synthesize the design and generate the FPGA configuration file, or to simulate the system.

B. Interfaces for IoT

Conversion between binary-radix and stochastic bitstreams is the major limitation in interfacing typical digital systems. Even tough it offers many parallel operators there are not many inputs available.

Connecting the SD from the rest of the supporting elements allows to integrate it in other systems, capable of interfacing with stochastic bitstreams, such as [10]. In this work the authors have created a cyber-physical system which interfaces analog sensors and actuators without the need to have either analog to digital and binary-to-stochastic converters, to acquire input data; and stochastic-to-binary and digital to analog converters to drive the actuators.

In essence, the generation of a bitstream from a binary-radix value requires more resources than an analog interface, but the analog interface requires a dedicated input pin.

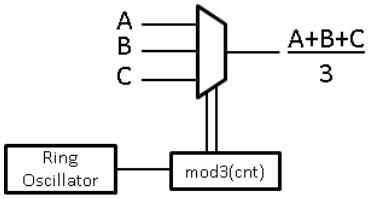

Fig. 10. Example of a 3-input stochastic adder with a STRO to minimize correlation between the bitstreams.

C. State-of-the-art Attributes

To hold on to the novel advancements in SC, the framework already includes a few research novelties to demonstrate its adaptability. The incorporated features which mitigate some of the limitations in SC and improve the quality of the results

1) Burn-in Period: There are stochastic arithmetic units are based on FSM, thus the values at their outputs are not instantly produced. Therefore, to account for such units, a counter is included to introduce enough latency. The outputs become valid once the counter reaches the threshold value.

2) Independent and Uncorrelated Units: As mentioned previously, correlation between bitstreams leads to weak results. To reduce such correlations [9] introduced STROs to generate spread-spectrum, individual and uncorrelated clock sources for each synchronous stochastic unit. Moreover, the authors also claim reduction in the power dissipated in the clock trees, without the penalty of introducing synchronizers, or alternative components, typical of asynchronous circuit designs [16]. This feature, which consists of a configurable length ring oscillator can be instantiated to provide the clock signal to all synchronous components in a SD, as illustrated in Fig. 10.

D. Simulation Platform

To facilitate the system development and verification, the proposed framework supports both RTL and gate-level simulation. This functionality is granted by a VHDL top-level

entity, automatically and specifically created for each system. This top-level entity automatically interfaces the generated SD, by connecting its inputs and outputs. Regarding the input stimulus for the simulation, it uses the values from the problem specification. The same information is then used to validate the obtained results at the end of the simulation.

One of the flagships of stochastic computing is its inherent resilience to faults. Doing fault-tolerance tests requires a rather complicated laboratory environment or native support from the FPGA to change configuration bits according to existent models. Therefore, to alleviate the designer from such, the proposed framework supports simulation of the complete system using fault-injection models.

The actual simulation process itself is supported directly by Modelsim, through the execution of a custom script which compiles all source files, executes the simulation and displays the waveforms for all signals. Figure 11 depicts the waveforms for some of the signals from a simulation of a stochastic system. It is worth noticing the buses which aggregate and organize the bitstreams in the design.

E. Fault Injection

Fault injection is performed at RTL level on the SD or the complete system, and they can be configured to upset the system as transient or permanent faults. In either case the faults injected are stuck-at 0 or 1 faults.

The fault injection is supported only at the simulation level through the use of Modelsim scripts. Each fault is characterized by an identifier of a net from the circuit, simulation time of occurrence and logic level of the fault. All faults are generated by a Python script before running the simulation following a specific probability distribution, e.g. Weibull or Normal distributions.

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

F. Evaluation Platform

The proposed framework provides a test platform to run any SD generated by it on an FPGA. It creates a fully functional autonomous stochastic system, containing the SD derived from the mathematical expression. The system supports SD of any size, being limited by the resources available on the FPGA device. This test platform manages the input and output signals required by the SD, along with the required conversions to be accessed by the host computer. Figure 12 depicts the system to be implemented on the FPGA. On the edges there are the conversion blocks, and in the middle the unit corresponding to the SD.

1) Architecture: The test platform circuit is constituted by the circuit under test (i.e. a simple arithmetic unit or a SD), the bitstream generators, and the output calculators.

It includes the units for the generation of the stochastic bitstreams from binary values previously stored in Block Random Access Memorys (BRAMs), and the result converters back to binary and its storage in other BRAMs. In more detail, each of these units supports many parallel bitstreams.

2) Process: The process of evaluating a SD starts with the configuration of the FPGA with the bitstream. Thereafter it is ready to exchange data with the host computer. The whole process is controlled by the host computer via Tool Command Language (TCL) scripts, and illustrated in fig. 13. The FSM controls the test process. It waits for the indication from the host computer to start the generation of the input bitstreams and starts counting the burn-in period, from binary values stored in BRAMs. After the burn-in period is over, the FSM starts the conversion of the output bitstreams.

IV. Karhunen-Loève Transform

A. Background

The KLT, also known as Principal Component Analysis (PCA), is an algorithm widely used in Machine Learning to reduce the dimensionality of data sets of many correlated variables, and is formulated as follows. Given a set of N data $x^i \in R^P$, where $i \in [1, N]$ an orthogonal basis described by a matrix Λ with dimensions $P \times K$ can be estimated that projects these data to a lower dimensional space of K dimensions. The projected data points are related to the original data through the formula in (6), written in matrix notation, where $X = [x^1, x^2, ..., x^N]$ and $F = [f^1, f^2, ..., f^N]$, where $f^i \in R^K$ denote the factor coefficients.

$$F = \Lambda^T X. \tag{6}$$

The original data is described from the lower dimensional space via (7):

$$X = \Lambda F + D \tag{7}$$

where D is the error of the approximation. The objective of the transform is to find a matrix Λ that has the Mean-Square Error (MSE) of the data approximation minimized. A standard technique is to evaluate the matrix Λ iteratively as described in steps (8) and (9), where λ_j denotes the j^{th} column of the Λ matrix.

$$\lambda_j = arg\ max\ \mathrm{E}\{(\lambda_j^T X_{j-1})^2\} \tag{8}$$

$$X_j = X - \sum_{k=1}^{j-1} \lambda_k \lambda_k^T X \tag{9}$$

where $X = [x^1 x^2 ... x^N]$, $X_0 = X$, $\|\lambda_j\| = 1$ and $\mathrm{E}\{.\}$ refers to expectation.

Fig. 14 presents a sample of the input data set. The data set consists of 68 images of 20x25 pixels. To compress the images from their dimension of 500 pixels down to 100 pixels, the λ matrix is required to have 500x100 elements. This application resembles the image compression in surveillance systems.

B. Implementation and Results

The KLT algorithm is based on the dot-product operation, which can be implemented using different circuits. Figure 15 shows the datapath for: a) rolled and b) unrolled architectures of a dot-product based circuit, to implement the datapath of one projection vector from a Z^p to Z^k KLT.

The circuit receives data from the input stream, identified with X. The samples, from the input stream, for each dimension p, are multiplied by the corresponding projection vector

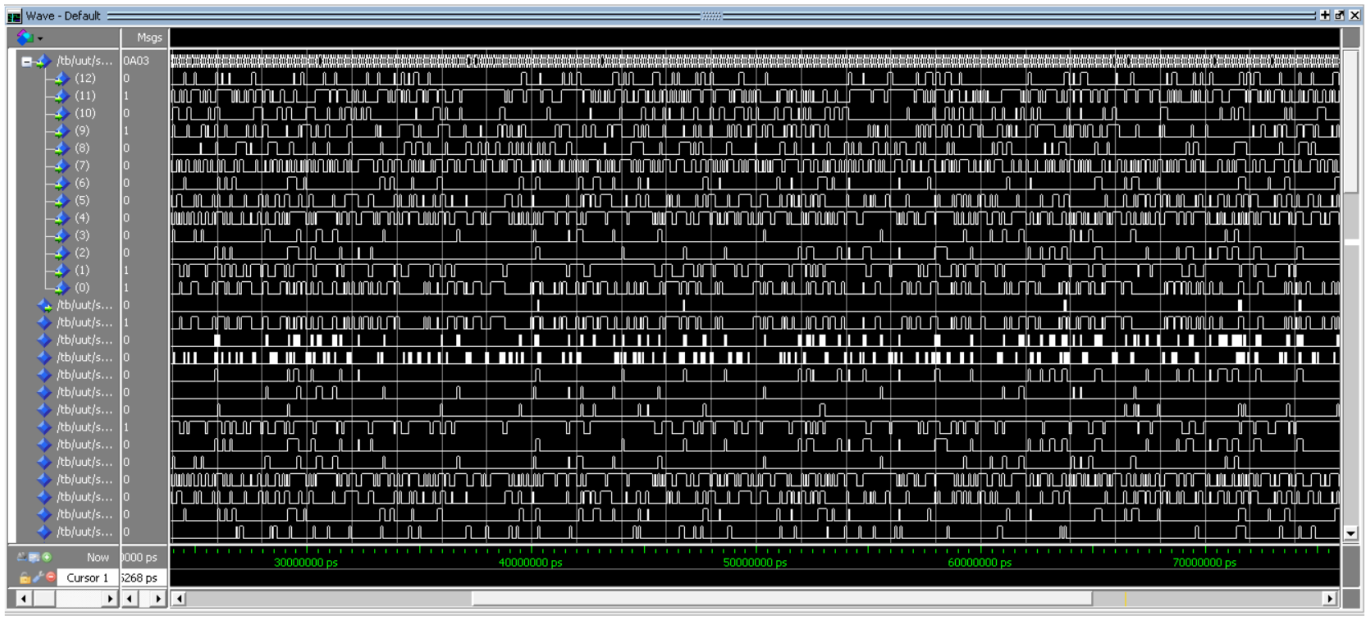

Fig. 11. Waveforms of a stochastic system Modelsim simulation.

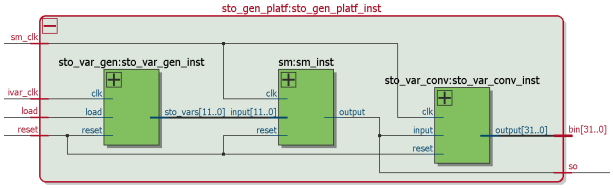

Fig. 12. RTL of a test circuit for a Stochastic Datapath.

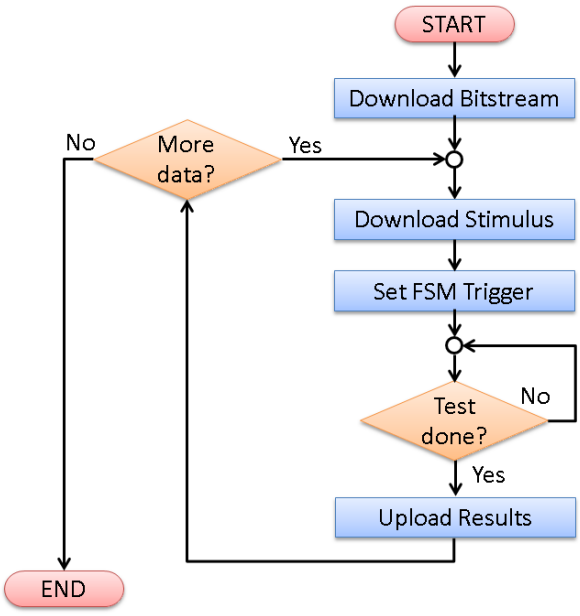

Fig. 13. Actions performed in the test process.

Fig. 14. Faces used as input data for the KLT designs.

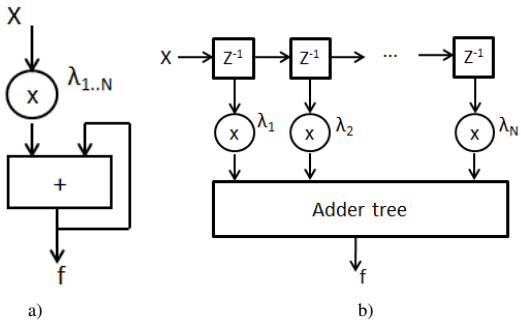

Fig. 15. Schematics of the datapath of a dot-product, for a projection vector of a KLT circuit: a) rolled and b) unrolled architectures.

λ_{pk}. The output of the multiplier is connected to an adder to do the accumulation. The final result is placed in the output stream, identified with f_k.

In this experiment, to compare binary-radix against SC, it was considered the unfolded architecture, which is the one that maximizes the parallelism offered by FPGAs. Considering also 9-bit binary-radix representation, it corresponds to a 512-bit bitstream. However, the length of the bitstream has no influence on the SD.

The implementation of the aforementioned KLT example in a complete parallelized would require to compute 100 streams with 500 multiply-accumulate operations, thus it is necessary to evaluate what is the maximum level of parallelism, and determine if the adoption of the SC is the most favorable approach.

The introduction of the expression for the KLT is generated in Python to explore different possible implementations for the problem via the following script:

```
[frame=single]
inp = 0
```

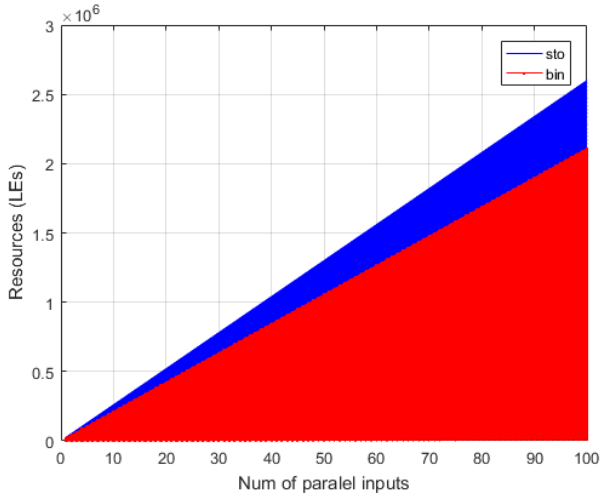

Fig. 16. Resources required to implement KLT using binary-radix (red) and SC (blue) for different numbers of input and parallel streams, with radix conversion units.

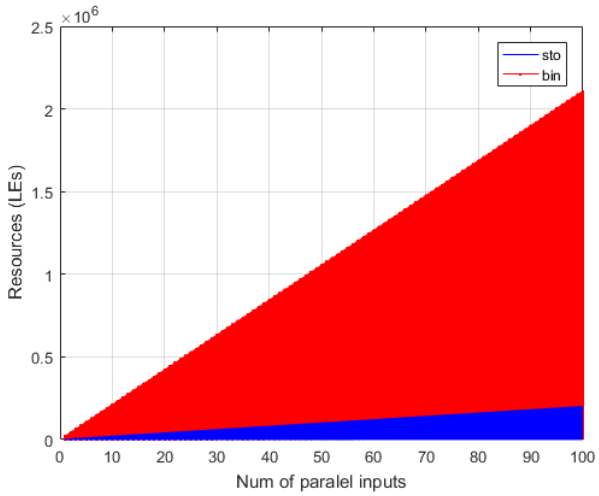

Fig. 17. Resources required to implement KLT using binary-radix (red) and SC (blue) for different numbers of input and parallel streams, without radix conversion units.

```
outp = []
expr = []
mac_inputs = 2
num_streams = 10

for i in range(0,num_streams):
    for j in range(0,10,mac_inputs):
        for k in range(0,mac_inputs):
     outp.append("in" + str(inp))
     inp = inp + 1
        outp.append("*")
    outp.append("+")
expr.append(outp)
```

Figure 16 presents the comparison of the resources required by both types of implementation using different numbers of inputs and parallel streams. For the SD implementation,on the left, the inputs are associated with converters, which penalizes the solution by requiring 18% extra resources. The plot on the right shows the results for the same implementation but without considering the conversion of the inputs, leading a SD solution which in the worst case requires 10% of the resources for the binary-radix solution. The results for the synthesis, in terms of resources, of the binary-radix and SC were modeled, using a linear approximation, to reduce the number of synthesis required to perform the evaluation.

V. Analysis

A. Design Automation

The present work could be of benefit for the engineer that is not familiarized with SC and is considering adopting it to use the processing power of a sensor node and improve the reliability of the system. By automatically producing and evaluating traditional datapaths in their SC implementations, not only is possible to compare resources but also evaluate its performance when operating variation of the operating conditions (power, temperature).

B. Bitstream Time Overhead

In SC each value is encoded as a bitstream over time instead of a parallel set of bits at one. Therefore, in the case-study presented the latency to produce a valid computation is given by the clock cycles to reach the length of the projection vector plus the time to perform multiplication and go through the adder tree, which is given by equation 10:

$$T = P_{ProjLen} + T_{Mult} + T_{AdderTree} \tag{10}$$

For both cases $P_{ProjLen}$ is the same as the number of inputs. However for the particular case of SC, T_{Mult} requires 2^{WL} clock cycle and so does $T_{AdderTree}$, but delayed by one clock cycle. For typical parallel binary T_{Mult} is 1, considering a fully combinatorial multipliers, and $T_{AdderTree}$ is $log_2(ProjLen)$.

The tradeoff is given in terms of the size of the projection vector and the wordlength (WL) adopted. Thus, the latency SC implementation would only produce results faster if $log_2(ProjSize)$ is greater than 2^{WL}. A parallel binary system is able to produce a result per clock cycle, whereas an SC system requires 2^{WL} clock cycles.

VI. Conclusions

This work introduces Stochastic Theater, a framework to specify, simulate, synthesize and test SDs on FPGAs, targeting IoT devices. It combines the bit-level specification for the processing of the stochastic bitstreams, and massive parallelization supported by the logic elements existent on FPGAs. The proposed framework also introduces support for simulation of stochastic systems when in the presence of faults. This paper also presents an evaluation of the proposed framework by producing system designs to implement different arithmetic expressions. Future work involves including support for Process, Voltage and Temperature (PVT) variation as in [6] to enable further research low-power designs for Stochastic Computing. This stochastic framework has been implemented mainly in Python and VHDL.

ACKNOWLEDGMENT

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references UID/CEC/50021/2013 and PTDC/EEI-HAC/31819/2017. The authors would like to thank Altera University Program for the donation of the FPGA board.

REFERENCES

[1] S. J. Agate and C. G. Drury. Electronic calculators: which notation is the better? *Applied Ergonomics*, 11(1):2 – 6, 1980.

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

[3] Peter Alfke. Efficient shift registers, lfsr counters, and long pseudo-random sequence generators, July 1996.

[4] B.D. Brown and H.C. Card. Stochastic neural computation. i. computational elements. *Computers, IEEE Transactions on*, 50(9):891–905, Sep 2001.

[5] Yun-Nan Chang and K.K. Parhi. Architectures for digital filters using stochastic computing. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 2697–2701, May 2013.

[6] R. P. Duarte and C. Bouganis. Arc 2014 over-clocking KLT designs on FPGAs under process, voltage, and temperature variation. *ACM Trans. Reconfigurable Technol. Syst.*, 9(1):7:1–7:17, November 2015.

[7] R. P. Duarte, J. Lobo, J. F. Ferreira, and J. Dias. Synthesis of bayesian machines on FPGAs using stochastic arithmetic. 2nd International Workshop on Neuromorphic and Brain-Based Computing Systems (NeuComp 2015), associated with DATE2015, Design Automation Test Europe 2015, March 2015.

[8] R. P. Duarte and H. Neto. Stochastic processors on FPGAs to compute sensor data towards fault-tolerant IoT systems. In *Dependable and Secure Computing (DSC), 2018 IEEE Conference on*, Dec 2018.

[9] R. P. Duarte, M. Vestias, and H. Neto. Enhancing stochastic computations via process variation. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 519–522, Aug 2015.

[10] R. P. Duarte, M. Vestias, and H. Neto. Xtokaxtikox: A stochastic computing-based autonomous cyber-physical system. In *Proceedings of the 1st IEEE International Conference on Rebooting Computing (ICRC)*, 2016.

[11] B. R. Gaines. Techniques of identification with the stochastic computer. In *in "Proc. International Federation of Automatic Control Symposium on Identification, Progue*, 1967.

[12] B.R. Gaines. Stochastic computing systems. volume 2, page 37, 1965.

[13] D. M. KASPRZYK, C. G. DRURY, and W. F. BIALAS. Human behaviour and performance in calculator use with algebraic and reverse polish notation. *Ergonomics*, 22(9):1011–1019, 1979.

[14] P. Li, D. J. Lilja, W. Qian, M. D. Riedel, and K. Bazargan. Logical computation on stochastic bit streams with linear finite-state machines. *IEEE Transactions on Computers*, 63(6):1474–1486, June 2014.

[15] Mingjie Lin, Ilia Lebedev, and John Wawrzynek. High-throughput bayesian computing machine with reconfigurable hardware. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, pages 73–82, New York, NY, USA, 2010. ACM.

[16] A.J. Martin and M. Nystrom. Asynchronous techniques for system-on-chip design. *Proceedings of the IEEE*, 94(6):1089–1120, June 2006.

[17] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D.S. Modha. A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm. In *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, pages 1–4, Sept 2011.

[18] Nadia Nedjah and Luiza de Macedo Mourelle. Reconfigurable hardware for neural networks: binary versus stochastic. *Neural Computing and Applications*, 16(3):249–255, May 2007.

[19] Weikang Qian, Xin Li, M.D. Riedel, K. Bazargan, and D.J. Lilja. An architecture for fault-tolerant computation with stochastic logic. *Computers, IEEE Transactions on*, 60(1):93–105, Jan 2011.

[20] N. Saraf, K. Bazargan, D.J. Lilja, and M.D. Riedel. IIR filters using stochastic arithmetic. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6, March 2014.

[21] M. Suri, O. Bichler, D. Querlioz, G. Palma, E. Vianello, D. Vuillaume, C. Gamrat, and B. DeSalvo. Cbram devices as binary synapses for low-power stochastic neuromorphic systems: Auditory (cochlea) and visual (retina) cognitive processing applications. In *Electron Devices Meeting (IEDM), 2012 IEEE International*, pages 10.3.1–10.3.4, Dec 2012.

[22] J. von Neumann. Probabilistic logics and synthesis of reliable organisms from unreliable components. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 43–98. Princeton University Press, 1956.

[23] Jieyu Zhao. *Stochastic Bit Stream Neural Networks*. Phd thesis, London University, 1995.

[24] Fan Zhou, Jun Liu, Yi Yu, Xiang Tian, Hui Liu, Yaoyao Hao, Shaomin Zhang, Weidong Chen, Jianhua Dai, and Xiaoxiang Zheng. Field-programmable gate array implementation of a probabilistic neural network for motor cortical decoding in rats. *Journal of Neuroscience Methods*, 185(2):299 – 306, 2010.