UpTime: Towards Flow-based In-Memory Computing with High Fault-Tolerance

Sven Thijssen[1], Muhammad Rashedul Haq Rashed[2], Sumit Kumar Jha[3], and Rickard Ewetz[4]
[1,3]*Department of Computer Science,* [2,4]*Department of Electrical and Computer Engineering*
[1,2,4]*University of Central Florida, Orlando, USA,* [3]*University of Texas at San Antonio, San Antonio, USA*
sven.thijssen@knights.ucf.edu, rashed09@knights.ucf.edu, sumit.jha@utsa.edu, rickard.ewetz@ucf.edu

***Abstract*—Processing in-memory promises to accelerate data-intensive applications by breaking von-Neumann based design principles. Flow-based computing is an in-memory computing paradigm that has shown immense potential for executing Boolean logic. Unfortunately, the immature fabrication processes for nanoscale memristor crossbars still struggle with yield challenges and run-time defects, which may render the computing system non-functional. Even worse, no previous studies have investigated the fault-tolerance of flow-based computing systems, which could potentially limit the capabilities of the entire paradigm. In this paper, we propose the UpTime framework to provide guaranties on the functional correctness and to maximize the lifetime of flow-based computing systems. The framework utilizes data layout organization to mitigate errors from faults with known type and location. To handle defects occurring at run-time, we propose the use of an error detection signal that can be evaluated with low overhead. The experimental evaluation demonstrates that the UpTime framework is capable of guaranteeing functional correctness for an average of 15.24 years. The up-time to down-time ratio is 99.9992%. Compared with utilizing the state-of-the-art write-verify scheme, the proposed error signal reduces power consumption by 25% and increases throughput by 6%, respectively.**

I. Introduction

Traditional ubiquitous computer architectures using CMOS technology are being challenged by the end of Moore's law [22], the end of Dennard scaling [2], [6] and the von Neumann bottleneck [1]. In-memory computing using nanoscale memristor crossbars poses promising results in terms of power consumption and latency [17], [20]. The advantages arise from eliminating the power-hungry and bandwidth-limited communication on the system bus.

The evaluation of Boolean functions has been investigated based on logic families such as IMPLY [12], MAGIC [11], and flow-based computing [9]. While MAGIC has shown great potential for accelerating applications based on matrix-vector multiplication [16], flow-based computing has shown to be advantageous for executing general-purpose Boolean logic [19]. Flow-based computing is based on first synthesizing a Boolean function ϕ into a crossbar design $\mathcal{D}$. The crossbar design specifies the mapping of Boolean variables to the memristors in the crossbar. To evaluate the Boolean function ϕ, the devices in the crossbar are programmed to be on/off with respect to a Boolean input vector. Next, the Boolean function ϕ is evaluated by merely observing if there is a flow of current from the input to the output of the crossbar.

This work was in part supported by NSF awards CCF-2113307, and CNS-1908471.

A key challenge for every in-memory computing paradigm is the presence of defects and environmental variations. Digital in-memory computing paradigms have inherent resilience to voltage fluctuation, noise, and temperature variations due to the separation of the logic one and zero [4]. Nevertheless, non-volatile memory devices can suffer stuck-at-fault defects which can have an detrimental impact on the functional correctness [10]. A stuck-at-fault defect is when a non-volatile memory device becomes stuck to high or low resistance and cannot be further programmed. The stuck-at-fault defects can occur at the time of fabrication or at run-time from heavy device utilization. Significant improvements have been made to the fabrication processes in recent years, resulting in crossbars with low defect rates (0.2% in [14]) and improved endurance. Unfortunately, a single stuck-at-fault defect can corrupt the correctness of an entire computing system. To detect write errors, a write-verify scheme can be employed [18]. In this scheme, the resistive state of a memristor is verified using a read operation. If there is a discrepancy between the actual resistance and the intended resistance, the device resistance is programmed in a series of fine-grained write operations.

While noteworthy efforts have been devoted to investigate automated synthesis techniques for flow-based computing [3], [19], no previous work have investigated the fault-tolerance of flow-based computing systems. However, fault-tolerance has been widely investigated for approximate analog in-memory matrix-vector multiplication. The common themes of handling stuck-at-faults are based on hardware redundancy [24], neural network retraining [7], and data layout organization [26]. It is appealing to adapt these techniques for flow-based computing systems. However, those schemes rely on that the underlying applications only require approximate precision. Consequently, one needs to fundamentally rethink how to apply the techniques for flow-based computing systems.

In this paper, we propose the UpTime framework to guarantee the functional correctness of flow-based computing systems to fabrication defects and run-time defects. The objective is to maximize system life-time and minimize down-time. The framework first applies data layout organization using an ILP formulation to mask the defects introduced at the time of fabrication. Next, during the execution, a low-overhead error detection signal is used to detect any run-time defects to avoid using write-verify. When an error is detected, the data layout organization technique is again invoked to mask the defect. The experimental evaluation shows that UpTime guarantees functional correctness for an average of 15.24 years. The up-

time to down-time ratio is 99.9992% on average. Compared with the state-of-the-art write-verify scheme in [18], power consumption and throughput are improved by 25% and 6%.

The paper is organized as follows: background in Section II and problem formulation in Section III. In Section IV, we introduce the UpTime framework. In Section V, we describe the data layout organization method. The error detection signal is explained in Section VI. The experimental evaluation is provided in Section VII and conclusions in Section VIII.

II. Background

A. Nanoscale memristor crossbars

A nanoscale memristor crossbar consists of two layers of nanowires. Each layer has parallel nanowires and layers are perpendicular to one another. At the intersections of the nanowires, a memristor connects two nanowires. The resistance of the memristor can be programmed to either a low (ON) or high (OFF) resistive state. The behaviour of a memristor is similar to the behaviour of a switch in the sense that a memristor in low resistive state can be considered a closed switch, and a memristor in high resistive state can be considered an open switch.

B. Flow-based computing

Flow-based computing is an in-memory computing paradigm, consisting of two phases: an initialization phase and a evaluation phase [19]. In the initialization phase, a crossbar design $\mathcal{D}$ is constructed for a Boolean formula ϕ by assigning Boolean variables, the negation of the Boolean variables, and the Boolean values true and false to the memristors. In Figure 1(a), a crossbar design $\mathcal{D}$ is illustrated for the Boolean formula $\phi = a \vee (\neg a \wedge b \wedge c \wedge d)$. Here, $\{a, b, c, d\} \cup \{\neg a, \neg b, \neg c, \neg d\} \cup \{0, 1\}$ are the literals and truth values to be assigned to the memristors. In the evaluation phase, the crossbar design is instantiated with an input vector (a=0,b=1,c=1,d=1) by programming the memristors to their respective states, as illustrated in Figure 1(b). Computation is performed by applying a high input voltage at the bottom-most nanowire (input) of the crossbar. Whenever the electrical current flows through the memristor crossbar to the top-most nanowire, the function ϕ evaluates to true. Otherwise, the function ϕ evaluates to false. In Figure 1(c), the path is illustrated by red arrows.

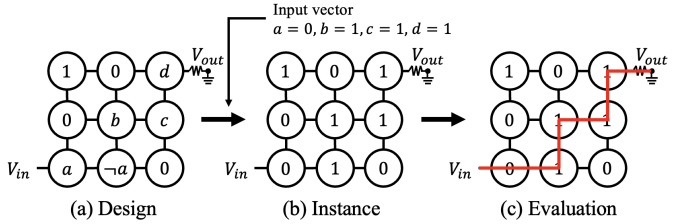

Fig. 1. Computation using flow-based computing.

The state-of-the-art automated synthesis methods for flow-based computing are based on binary decision diagrams (BDDs) [3], [19]. The BDD can be represented using a directed acyclic graph (DAG). In [19], the nodes in the DAG are assigned to the wordlines and bitlines of the memristor crossbar, and the edges in the DAG are assigned to the memristors.

C. Stuck-at-faults

A stuck-at-fault denotes the situation when a memristor cannot be further programmed. Two types of stuck-at-faults exist: stuck-on and stuck-off faults. The former type results in devices with low-resistive state, and the latter results in devices with high resistive state [15]. The stuck-at-fault can be introduced during fabrication, or during device utilization resulting in runtime defects. The runtime defects occur from the continuous on/off switching of the memristors [24].

These stuck-at-faults introduce unwanted behavior for flow-based computing, which can potentially result in incorrect evaluations. Consider the crossbar design $\mathcal{D}$ in Figure 2(a) and the crossbar X capturing the state of the hardware in Figure 2(b). In the crossbar X, the stuck-off, stuck-on, and non-defective memristors are denoted with (0), (1), and $(-)$, respectively. The defective cells are also shown with a dashed border. Mapping the design $\mathcal{D}$ to the crossbar X forms the effective crossbar design $\mathcal{D}_X$, which is shown in Figure 2(c). The effective crossbar design $\mathcal{D}_X$ is obtained by taking the stuck-at-faults in X and the values for the remaining memristors from $\mathcal{D}$. It can be observed that the stuck-on fault is masked by the design whereas the stuck-off fault is not masked. Consider the input vector a=0, b=1, c=1, and d=1. The crossbar design $\mathcal{D}$ evaluates to true and the effective crossbar design $\mathcal{D}_X$ evaluates false, i.e., the unmasked stuck-off fault has compromised the functional correctness.

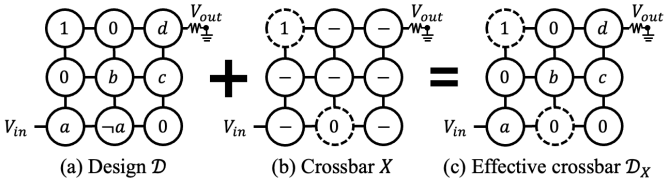

Fig. 2. (a) A crossbar design $\mathcal{D}$. (b) The crossbar X, where '0', '1', '-' indicate stuck-off, stuck-on, and functional memristors, respectively. (c) The effective crossbar design $\mathcal{D}_X$. $\mathcal{D}$ in (a) and $\mathcal{D}_X$ in (c) are not functionally equivalent, and the effective crossbar design produces incorrect outputs when evaluating the input vector a=0, b=1, c=1, d=1.

III. Problem Definition

This paper aims to develop fault-tolerance techniques for flow-based computing systems. The goal is to guarantee functional correctness and to maximize the computing system lifetime while minimizing downtime. Towards this overall objective, we solve two key subproblems:

1) Handling stuck-at-faults with known type and location.
2) Detection of stuck-at-fault introduced at runtime.

To solve the first subproblem, we observe that any two rows (or columns) in a crossbar design can be reordered without changing the functionality of the crossbar design. Hence, there exist an opportunity to reorder the rows and columns using data layout organization to mask all the known defects in the hardware. Note that we do not reorder the input and output rows in our implementation. To solve the second subproblem, we propose to leverage an error detection that efficiently computes $\neg\phi$. If ϕ and $\neg\phi$ are equal, it can be determined that an error has occurred. This eliminates the need to verify each write operation in the evaluation process.

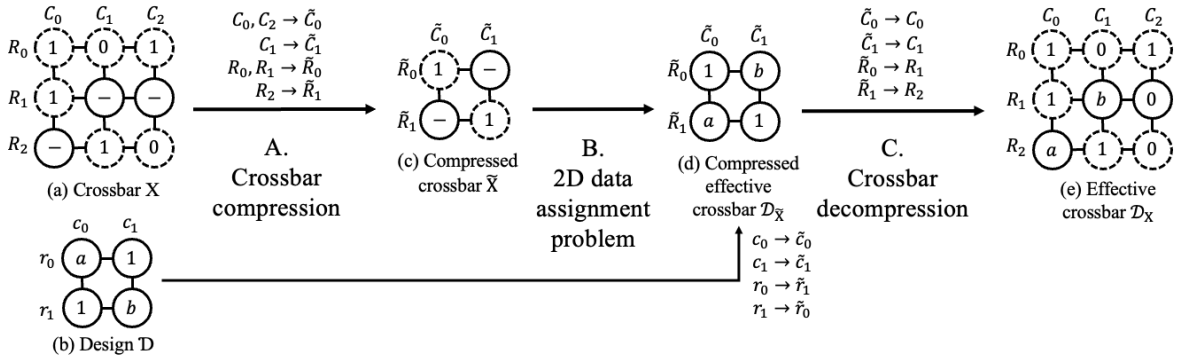

Fig. 4. Data layout organization consists of three steps: crossbar compression, 2D data assignment problem, and crossbar decompression. In the first step, the crossbar X of (a) is compressed into an equivalent compressed crossbar $\tilde{X}$ of (c). Then, the crossbar design $\mathcal{D}$ of (b) is assigned to the compressed crossbar $\tilde{X}$ such that we obtain the compressed effective crossbar $\mathcal{D}_{\tilde{X}}$ of (d). Finally, $\mathcal{D}_{\tilde{X}}$ is decompressed into an effective crossbar $\mathcal{D}_X$ as in (e).

IV. UpTime framework

In this section, we introduce the UpTime framework. An overview is given in Figure 3. The input of the framework is a design $\mathcal{D}$ for a Boolean formula ϕ and the crossbar hardware obtained from fabrication. First, the crossbar X is obtained by identifying the type and location of the stuck-at-faults introduced at fabrication using squeeze-search scheme [5]. Then, data layout organization is performed to mask all the defective memristors in X using the data $\mathcal{D}$. The data layout organization is performed by reordering rows and columns in $\mathcal{D}$. The details of our proposed data layout reorganization method are discussed in Section V. Next, program execution is started up and performed until an error detection mechanism identifies a run-time stuck-at-fault. In Section VI, we introduce a low-overhead error detection mechanism based on evaluating both ϕ and $\neg\phi$. It turns out that $\neg\phi$ can be evaluated extremely efficiently due to the underlying properties of flow-based computing and nanoscale crossbars. Subsequently, the program is halted when an error is detected, and the stuck-at-faults are again localized and identified to then restart the program using a new data layout reorganization. The execution continues until the stuck-at-faults can no longer be masked.

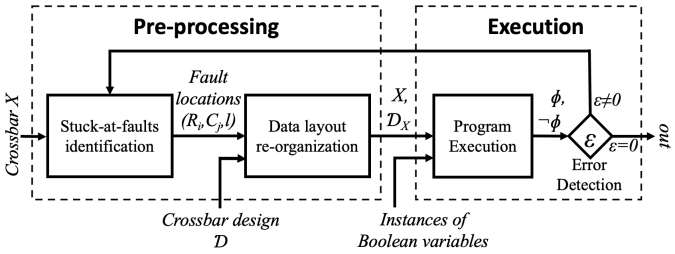

Fig. 3. Overview of the UpTime framework.

V. Data layout organization

In this section, we propose a data layout organization technique to mask defects with known type and location. The proposed technique is illustrated with an example in Figure 4. The input is the crossbar X and the design $\mathcal{D}$, as shown in Figure 4(a) and Figure 4(b). The crossbar X may be slightly larger than $\mathcal{D}$ as crossbars are typically fabricated with pre-defined dimensions such as 128x128. The objective is to find an effective crossbar $\mathcal{D}_X$ that is functionally equivalent to $\mathcal{D}$ and masks all stuck-at-faults in X, as shown in Figure 4(e).

The technique consists of three steps: crossbar compression, 2D data assignment problem, and crossbar decompression. The first step compresses the crossbar X to $\tilde{X}$ based on stuck-on defects. In the second step, a 2D data layout organization problem is formulated and solved to obtain $\mathcal{D}_{\tilde{X}}$ in the compressed space. In the third step, decompression is performed to obtain $\mathcal{D}_X$. In the following subsections, we explain the details of these three steps.

A. Crossbar compression

Stuck-on faults introduce equivalences within the crossbar. More specifically, in flow-based computing, two nanowires are *logically equivalent* if they share a memristor in the low-resistive state. Thus, when two or more stuck-on faults are present in the same row (column), these rows (columns) represent the same Boolean function. For example, in Figure 4(a), we observe that the memristors at position (R_0, C_0) and (R_1, C_0) in crossbar X are both stuck-on faults. Consequently, the pair of rows (R_0, R_1) are functionally equivalent. Therefore, we propose to compress the two rows into an equivalent row. No solution space is sacrificed by the compression because no pair of rows (or columns) in a crossbar design $\mathcal{D}$ are equivalent[1].

Two equivalent rows/columns i and j can be merged into one new row/column by applying compression rules to all pairwise memristors in these rows/columns. The compression rules are provided in Table I. For example, in the crossbar X in Figure 4(a), the memristors in row R_0 and R_1 are column-wise compressed as follows: $(1, 1)$ into 1, $(0, -)$ into $-$, and $(1, -)$ into 1. Now, since the compressed row contains stuck-on memristors at position (R_0, C_0) and (R_0, C_2), the columns C_0 and C_2 are compressed, resulting in a 2x2 compressed crossbar $\tilde{X}$ in Figure 4(c).

TABLE I
Compression rules for a defective crossbar. Stuck-on faults, stuck-off faults and non-defective memristors are denoted by 1, 0 and −, respectively.

Row/column i	0	0	0	1	1	1	−	−	−
Row/column j	0	1	−	0	1	−	0	1	−
Compressed row/column	0	1	−	1	1	1	−	1	−

[1]If a pair of rows/columns were equivalent, you could simply remove one of the rows/columns and reduce the size of the design by one row/column.

B. 2D data assignment problem

In this section, we formulate and solve a 2D data assignment problem. The input is the crossbar design $\mathcal{D}$ and the compressed crossbar $\tilde{X}$. The problem is formulated to assign each row (column) in $\mathcal{D}$ to a row (column) in $\tilde{X}$. The objective is to perform the assignment while minimizing the number of non-masked defects.

We introduce the variable $r^{i,k} \in [0,1]$ for each row i in $\mathcal{D}$ and each row k in $\tilde{X}$. If the Boolean variable is equal to one, then row i in $\mathcal{D}$ is assigned to row k in $\tilde{X}$. Analogously, we introduce the variables $c^{j,l} \in [0,1]$ for the columns. Let E_{on} denote the errors arising from mapping the Boolean value 0 or a Boolean literal in $\mathcal{D}$ to a stuck-on fault in $\tilde{X}$. Let E_{off} denote the errors arising from mapping the Boolean value 1 or a Boolean literal in $\mathcal{D}$ to a stuck-off fault in $\tilde{X}$. The collective set of errors E is the union of both E_{on} and E_{off}: $E = E_{on} \cup E_{off}$. To capture the errors, we introduce the variables $e^{i,j,k,l} \in [0,1]$ for each error in E. An ILP formulation minimizing the errors is formulated, as follows [25]:

$$\begin{aligned}
\min \quad & \sum e^{i,j,k,l} \\
\text{s.t.} \quad & e^{i,j,k,l} \geq r^{i,k} + c^{j,l} - 1, && \forall (i,j,k,l) \in E \quad (1)\\
& e^{i,j,k,l} \leq r^{i,k}, && \forall (i,j,k,l) \in E \quad (2)\\
& e^{i,j,k,l} \leq c^{j,l}, && \forall (i,j,k,l) \in E \quad (3)\\
& \sum_{k \in R_{\tilde{\mathrm{X}}}} r^{i,k} = 1, && \forall i \in R_{\mathcal{D}} \quad (4)\\
& \sum_{l \in C_{\tilde{\mathrm{X}}}} c^{j,l} = 1, && \forall j \in C_{\mathcal{D}} \quad (5)\\
& \sum_{i \in R_{\mathcal{D}}} r^{i,k} \leq 1, && \forall k \in R_{\tilde{\mathrm{X}}} \quad (6)\\
& \sum_{j \in C_{\mathcal{D}}} c^{j,l} \leq 1, && \forall l \in C_{\tilde{\mathrm{X}}} \quad (7)
\end{aligned}$$

The first three constraints 1, 2, and 3 force $e^{i,j,k,l}$ to be 1 when both $r^{i,k}$ and $c^{j,l}$ are 1, indicating that when mapping both row i to row k and column j to column l introduces an error. Otherwise, when $r^{i,k} = 0$ or $c^{j,l} = 0$, then $e^{i,j,k,l}$ is forced to be 0. More specifically, the first three constraints represent a logical AND operation over $r^{i,k}$ and $c^{j,l}$. The constraints on lines 4 and 5 ensure that each row/column in the crossbar design $\mathcal{D}$ is assigned to exactly one row/column in the compressed crossbar $\tilde{X}$. The constraints on lines 6 and 7 ensure that at most one row/column from the crossbar design $\mathcal{D}$ is assigned to a row/column in the compressed crossbar $\tilde{X}$. In practice, we also do not reorder the input and output row, as they are fixated in hardware. The input of the data assignment problem are the compressed crossbar $\tilde{X}$ in Figure 4(c) and the design $\mathcal{D}$ in Figure 4(b), the output is the compressed effective crossbar $\mathcal{D}_{\tilde{X}}$ in Figure 4(d).

C. Crossbar decompression

The last step is to decompress the effective compressed crossbar $\mathcal{D}_{\tilde{X}}$ to the uncompressed space. The compression step can be viewed as a function $\mathcal{F}$ that maps a row R_i (column C_i) of the crossbar X to a row $\tilde{R}_i$ (column $\tilde{C}_i$) of the compressed crossbar $\tilde{X}$. Hence, to decompress the crossbar, one must apply the reverse of the function $\mathcal{F}$, i.e. $\mathcal{F}^{-1}$. However, $\mathcal{F}^{-1}$ maps a row $\tilde{R}_i$ (column $\tilde{C}_i$) of the compressed crossbar $\mathcal{D}_{\tilde{X}}$ to a set of rows $\{R_i\}$ (columns $\{C_i\}$) of the crossbar X. Consequentially the rows $\tilde{R}_i$ (columns $\tilde{C}_i$) of the compressed effective crossbar $\mathcal{D}_{\tilde{X}}$ map to the rows $\{R_i\}$ (columns $\{C_i\}$) of the effective crossbar $\mathcal{D}_X$. To assign a memristor $\tilde{M}_{ij}$ of the compressed effective crossbar $\mathcal{D}_{\tilde{X}}$ to a memristor M_{ij} in the effective crossbar $\mathcal{D}_X$, one chooses any memristor in the set of memristors $\{M_{ij} | \mathcal{F}^{-1}(\tilde{R}_i), \mathcal{F}^{-1}(\tilde{C}_j)\}$ that is less restrictive, i.e. to the same state of a memristor that is not a stuck-at-fault. The remaining memristors in the effective crossbar $\mathcal{D}_X$ are set to the low resistive state. The compressed effective crossbar $\mathcal{D}_{\tilde{X}}$ in Figure 4(d) is converted into the effective crossbar $\mathcal{D}_X$ in Figure 4(e).

VI. Error detection

In this section, we introduce an error mechanism for detecting defects introduced at run-time. The technique allows us to circumvent verifying each write operation using a read operation during the execution phase.

TABLE II
Error detection.

ϕ	$\neg\phi$	XOR(ϕ, $\neg\phi$)	Error detected?
0	0	0	yes
0	1	1	no
1	0	1	no
1	1	0	yes

The error detection technique is based on evaluating both the Boolean function ϕ (the specification) and its complement $\neg\phi$. ϕ and $\neg\phi$ should be complemented by definition. Using the XOR operation on the evaluation of ϕ and $\neg\phi$, one can detect whether this statement holds, as illustrated in Table II. If XOR(ϕ, $\neg\phi$) evaluates to false, there is at least one undetected stuck-at-fault defect that has been introduced at run-time in the crossbar. If XOR(ϕ, $\neg\phi$) evaluates to true, the output of the function could be correct or there could be multiple faults (at least 2) corrupting both ϕ and $\neg\phi$. However, we observe that it is *impossible* for single defect to simultaneously corrupt both ϕ and $\neg\phi$ at the same time. Therefore, as long as the error detection signal detects a run-time defect before the next fault is introduced, the system is guaranteed to maintain functional correctness. We observe in our experiments that any introduced defects are typically detected within a few cycles (less than 30) and the defects are substantially (orders of magnitude) more separated in time.

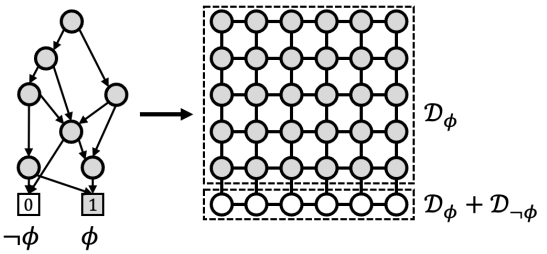

Fig. 5. BDD for ϕ and crossbar design for ϕ and $\neg\phi$.

Next, we turn our attention to implementing $\neg\phi$ cost efficiently in hardware. We propose to implement both ϕ and $\neg\phi$ in a single crossbar design. A BDD represents both ϕ and $\neg\phi$ in its DAG representation. If, for an input vector, there exists a path from the root node to the positive terminal node ($\top$/1), then the Boolean formula ϕ evaluates true. Otherwise, if, for an input vector, there exists a path from the root node to the negative terminal node ($\bot$/0), then the Boolean formula ϕ evaluates false, and thus $\neg\phi$ evaluates true. Remember that the nodes in the DAG representation of the BDD for a Boolean formula ϕ are assigned to wordlines and bitlines in the state-of-the-art mapping method [19]. Then, the hardware utilization increases slightly (more specifically by one wordline and possibly one bitline), given that the negative terminal node is now assigned to a wordline as well. In Figure 5(a), a BDD is given for a Boolean formula. In Figure 5(b), both ϕ and $\neg\phi$ are represented in a single crossbar design where the positive terminal node of the BDD is assigned to the second-bottom-most nanowire, and the negative terminal node of the BDD is assigned to the bottom most nanowire. When a high input voltage is applied to $V_{in,\phi}$, ϕ will be evaluated; when a high input voltage is applied to $V_{in,\neg\phi}$, $\neg\phi$ will be evaluated.

VII. Experimental evaluation

In this section, we will evaluate the proposed framework UpTime. The framework is implemented in Python 3.8 and the experiments are conducted on a machine with 20 Intel Core i9-9900X processors, running on Ubuntu 20.04.2. The framework is evaluated on benchmarks obtained from RevLib [21] and the crossbar designs are constructed following the algorithm in [19]. An overview of the benchmark specifications is provided in Table III. For the defective crossbars, we consider both hard faults due to fabrication errors, and soft faults due to incorrect write operations. Based on the experimental setup in [23], we model the fabrications faults based on multiple Gaussian distributions in the crossbar. By default, we set the number of defects to 0.2% [14]. The number of stuck-on vs stuck-off is set to $1:5$ [10]. The probability of a soft error occurring for a literal in the crossbar design at run-time is $p = 1 - e^{-\lambda \times \tau / 10^9}$ where λ is set to 10^6 and τ is the time in hours [13]. The power consumption and the latency for ReRAM write operation is $18nW$ and $100\mu s$ [8].

TABLE III
Dimensions for benchmarks.

Benchmark	Rows (num)	Column (num)
misex1	41	34
cm163a	68	61
5xp1	33	24
clip	41	45
cordic	58	58
frg1	25	21

In Section VII-A, we evaluate the proposed error detection signal. In Section VII-B, we evaluate the proposed UpTime framework, and we compare it with the state-of-the-art stuck-at-fault mitigation methods.

A. Evaluation of error detection signal

We compare our error detection signal with the state-of-the-art write-verify scheme [18] in Figure 6. Compared with write-verify, it can be observed that the UpTime framework reduces power consumption and latency with 25% and 6%, respectively. The area increases with 20% due to both ϕ and $\neg\phi$ being mapped. Both the increase in power consumption and latency of write-verify can be addressed to the peripheral circuitry and the additional read operations. Consequently, we use the proposed error detection signal as the default method for detecting stuck-at-fault defects in the UpTime framework.

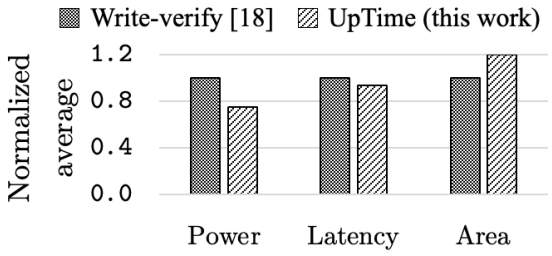

Fig. 6. Normalized average for power, latency, and area of a write-verify detection signal and our proposed detection signal.

B. Evaluation of UpTime framework

In this section, we will compare the accuracy for three different mapping schemes. The first scheme is a baseline method. In this scheme, the crossbar design is mapped to the crossbar without regard for any stuck-at-faults. The second scheme is row permutation after [27] and the third scheme is sequential row and column permutations after [23]. The fourth scheme is UpTime where stuck-at-faults are identified and mitigated at fabrication and at runtime using the proposed framework.

TABLE IV
Minimum, mean, and maximum lifetime for the schemes with row permutations only, sequential row and column permutations, and the proposed UpTime framework.

Benchmark	Scheme	Min (years)	Mean (years)	Max (years)
misex1	Baseline	0.00	0.00	0.00
	Row [27]	0.00	0.15	1.13
	Sequential row & column [23]	0.00	0.32	1.13
	UpTime (this work)	11.24	14.19	15.60
cm163a	Baseline	0.00	0.00	0.00
	Row [27]	0.00	0.00	0.00
	Sequential row & column [23]	0.00	0.06	0.29
	UpTime (this work)	8.08	9.10	10.75
5xp1	Baseline	0.00	0.00	0.00
	Row [27]	0.05	1.08	2.65
	Sequential row & column [23]	0.05	1.24	2.65
	UpTime (this work)	13.32	15.85	17.90
clip	Baseline	0.00	0.00	0.00
	Row [27]	0.00	0.01	0.08
	Sequential row & column	0.00	0.20	0.72
	UpTime (this work)	11.24	13.77	14.83
cordic	Baseline	0.00	0.00	0.00
	Row [27]	0.00	0.00	0.00
	Sequential row & column [23]	0.00	0.07	0.29
	UpTime (this work)	9.08	10.53	12.61
frg1	Baseline	0.00	0.00	0.00
	Row [27]	0.22	2.07	4.13
	Sequential row & column [23]	0.85	2.13	4.13
	UpTime (this work)	15.05	17.52	19.74

In Table IV, we show the minimum, mean, and maximum, system lifetime for each of the three schemes obtained from 10 Monte Carlo simulations. Here, failure is defined as the point in time when the accuracy drops below 100%, ending the system's lifetime. The accuracy is defined to be the number of correct input vectors/total number of input vectors. In the table, we observe that the lifetime for the baseline is lowest, indicating that flow-based computing is susceptible to stuck-at-faults without error mitigation. Then follows row permutations and, sequential row and column permutations. This is due to the fact that there may not be a feasible solution for row permutations, but there may be a feasible solution for column permutations, resulting in that the second scheme slightly increases lifetime compared to the first scheme. Finally, our proposed framework UpTime outperforms both of previous approaches due to the simultaneous exploration of row and column permutations.

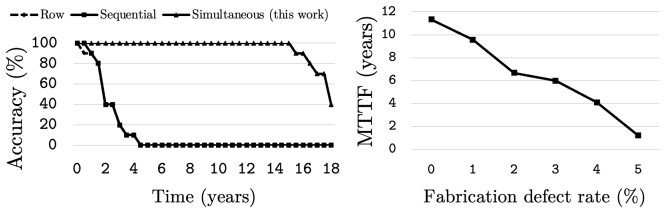

Fig. 7. (a) Average accuracy over 10 simulations for the row permutations after in [27], the sequential row and column permutations in [23], and the proposed framework UpTime. (b) Mean time to failure (MTTF) for different fabrication defect rates.

Finally, in Figure 7(a), we show the accuracy in terms of time for the benchmark *frg1* using the three schemes (except for the baseline method as its lifetime is insignificant). Supporting Table IV, it can be observed that the lifetime of the scheme using row permutations only results in a lifetime of less than a year. The lifetime based on sequential row and column permutations increases the lifetime to little over a year. However, it can be observed that the UpTime framework maintains 100% accuracy for more than fifteen years. In Figure 7(b), the mean time to failure (MTTF) is given in terms of the fabrication defect rate. It can be observed that even for fabrication defect rates as high as 5%, the system will approximately have a lifetime of one year.

TABLE V
MEAN TIME TO FAILURE, AVERAGE DOWNTIME, AND TOTAL UPTIME OF THE UPTIME FRAMEWORK FOR DIFFERENT BENCHMARKS.

Benchmark	Mean (years)	Average downtime (hours)	Uptime (%)
misex1	14.19	0.27	99.9995%
cm163a	9.10	0.54	99.9984%
5xp1	15.85	0.16	99.9997%
clip	13.77	0.36	99.9993%
cordic	10.53	0.52	99.9986%
frg1	17.52	0.12	99.9998%

In Table V, an overview is given of the average downtime (hours) and the uptime (percentage) of a system for different benchmarks. The downtime and uptime are computed in terms of the average number of years before the system fails. We can observe that the lowest uptime is 99.9984% and the highest uptime is 99.9998%.

VIII. SUMMARY AND FUTURE WORK

Guaranteeing the functional correctness of a computing system is crucial in many applications. Unfortunately, fault-tolerance for flow-based computing has not yet been explored. Therefore, we have introduced the UpTime framework that allows a flow-based computing system to live with 100% accuracy for several years. The framework consists of a low-overhead error detection signal and a data layout reorganization scheme. The error detection signal is based on the evaluation of a Boolean function and its negation whereas the data layout organization masks all stuck-at-faults in the crossbar.

REFERENCES

[1] J. Backus. Can programming be liberated from the von neumann style? *CACM*, 21(8):613–641, 1978.
[2] M. Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *SSCS Newsletter*, 12(1):11–13, 2007.
[3] D. Chakraborty and S. K. Jha. Automated synthesis of compact crossbars for sneak-path based in-memory computing. In *DATE*, pages 770–775. IEEE, 2017.
[4] S. Channamadhavuni et al. Accelerating ai applications using analog in-memory computing: Challenges and opportunities. In *GLSVLSI*, pages 379–384, 2021.
[5] C.-Y. Chen et al. Rram defect modeling and failure analysis based on march test and a novel squeeze-search scheme. *TC*, 64(1):180–190, 2014.
[6] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, et al. Design of ion-implanted mosfet's with very small physical dimensions. *JSSC*, 9(5):256–268, 1974.
[7] S. Jain et al. Rxnn: A framework for evaluating deep neural networks on resistive crossbars. *TCAD*, 40(2):326–338, 2020.
[8] K. Jeon, J. Kim, J. J. Ryu, et al. Self-rectifying resistive memory in passive crossbar arrays. *Nature communications*, 12(1):1–15, 2021.
[9] S. K. Jha et al. Computation of boolean formulas using sneak paths in crossbar computing, Apr. 19 2016. US Patent 9,319,047.
[10] G. Jung et al. Cost-and dataset-free stuck-at fault mitigation for reram-based deep learning accelerators. In *DATE*, pages 1733–1738. IEEE, 2021.
[11] S. Kvatinsky, D. Belousov, S. Liman, et al. Magic—memristor-aided logic. *TCAS-II*, 61(11):895–899, 2014.
[12] S. Kvatinsky et al. Memristor-based material implication (imply) logic: Design principles and methodologies. *VLSI*, 22(10):2054–2066, 2013.
[13] O. Leitersdorf, B. Perach, R. Ronen, and S. Kvatinsky. Efficient error-correcting-code mechanism for high-throughput memristive processing-in-memory. In *DAC*, pages 199–204. IEEE, 2021.
[14] C. Li, M. Hu, Y. Li, et al. Analogue signal and image processing with large memristor crossbars. *Nature electronics*, 1(1):52–59, 2018.
[15] M. Liu, L. Xia, Y. Wang, and K. Chakrabarty. Fault tolerance for rram-based matrix operations. In *ITC'18*, pages 1–10. IEEE, 2018.
[16] M. R. H. Rashed, S. K. Jha, and R. Ewetz. Hybrid analog-digital in-memory computing. In *ICCAD*, pages 1–9. IEEE, 2021.
[17] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, et al. Memory devices and applications for in-memory computing. *NNano*, 15(7):529–544, 2020.
[18] W. Shim et al. Two-step write-verify scheme and impact of the read noise in multilevel rram-based inference engine. *SST*, 35(11):115026, 2020.
[19] S. Thijssen et al. Compact: Flow-based computing on nanoscale crossbars with minimal semiperimeter. In *DATE'21*, pages 232–237. IEEE, 2021.
[20] N. Verma, H. Jia, H. Valavi, et al. In-memory computing: Advances and prospects. *SSC-M*, 11(3):43–55, 2019.
[21] R. Wille et al. Revlib: An online resource for reversible functions and reversible circuits. In *ISMVL 2008*, pages 220–225. IEEE, 2008.
[22] R. S. Williams. What's next?[the end of moore's law]. *Computing in Science & Engineering*, 19(2):7–13, 2017.
[23] L. Xia et al. Fault-tolerant training with on-line fault detection for rram-based neural computing systems. In *DAC'17*, pages 1–6, 2017.
[24] L. Xia, W. Huangfu, T. Tang, et al. Stuck-at fault tolerance in rram computing systems. *JETCAS*, 8(1):102–115, 2017.
[25] M. Zamani et al. Ilp formulations for variation/defect-tolerant logic mapping on crossbar nano-architectures. *JETC*, 9(3):1–21, 2013.
[26] B. Zhang and R. Ewetz. Towards resilient deployment of in-memory neural networks with high throughput. In *DAC'21*, pages 1–9, 2020.
[27] B. Zhang, N. Uysal, D. Fan, and R. Ewetz. Handling stuck-at-faults in memristor crossbar arrays using matrix transformations. In *ASP-DAC*, pages 438–443, 2019.