Verification of Flow-Based Computing Systems Using Bounded Model Checking

Sven Thijssen*, Suraj Singireddy†, Muhammad Rashedul Haq Rashed‡, Sumit Kumar Jha§, and Rickard Ewetz‡
*Department of Computer Science, University of Central Florida, Orlando, USA
†Department of Computer Science, University of Texas at San Antonio, San Antonio, USA
‡Department of Electrical and Computer Engineering, University of Central Florida, Orlando, USA
§Computer Science Department, Florida International University, Miami, USA
{sven.thijssen, muhammad.rashed, rickard.ewetz}@ucf.edu, suraj.singireddy@utsa.edu, jha@cs.fiu.edu

***Abstract*—Flow-based computing is a digital in-memory computing paradigm with tremendous potential. Its favorable characteristics, such as high robustness, low energy consumption and small computational delay make it a strong contender for integration into future computing systems. While most studies on emerging computing paradigms are focused on synthesis, it is crucial to develop methods to verify the functional correctness of the resulting designs. Flow-based computing is based on an undirected computational graph, which prevents equivalence checking to be performed by solving SAT formulations. In this paper, we propose a framework called XSAT for equivalence checking of crossbar designs for flow-based computing. The XSAT framework draws on bounded model checking (BMC) to convert the undirected computational graph into a directed acyclic computational graph (DAG). The conversion allows traditional SAT-based equivalence checking techniques to be used at the expense of increasing the size of the problem. We further introduce a divide-and-conquer technique to accelerate the verification process. The technique divides the main problem into many subproblems of smaller size, which can be executed in parallel using multiple cores or nodes. From the experimental evaluation, it can be observed that the XSAT framework can solve all nineteen MCNC benchmarks whereas previous SOTA techniques can only solve eleven out of the nineteen benchmarks within one hour, i.e., with speed-ups of one to two orders of magnitude. Moreover, the divide-and-conquer technique results in speed-ups of up to 93× on large benchmark circuits.**

I. Introduction

The emergence of big data has given rise to a plethora of applications that heavily rely on data for intelligent decision making. This includes applications within fields such as medicine [1], statistics [2], machine learning [3], [4], and deep neural networks [5], [6]. Unfortunately, the acceleration of data-intensive applications using today's high-performance computing systems is inhibited by the data transfer between memory and computing units [7]. This bottleneck –the von Neumann bottleneck– has ignited the field of in-memory computing, where the data transfer is mitigated by merging memory storage and computation on-chip [8], [9]. Over the years, in-memory computing paradigms such as analog matrix-vector multiplication [10], MAGIC [11], IMPLY [12], Bit-wise-in-bulk [13], and flow-based computing [14], [15] have been proposed. The different paradigms are advantageous for applications with different properties. Flow-based computing is a primary contender for efficiently accelerating Boolean functions [16], [17]. This stems from the underlying properties of (i) deterministic precision, (ii) high-speed, and (iii) in-situ computation [16], [18]. The deterministic precision is a result of the clear distinction between the two logic states zero and one. The high-speed comes from utilizing the flow of electrical currents for evaluating Boolean functions. The in-situ computation is enabled by compiling the desired functionality into a crossbar design prior to execution.

This work was in part supported by NSF awards # 2319399, # 2113307, the University of Central Florida, and Texas STARs award to Sumit Jha.

The flow-based computing paradigm consists of two phases. First, in the synthesis phase, a crossbar design is constructed to realize a Boolean function. This design is an assignment of Boolean literals and truth values to memristors. Second, during execution, the memristors are programmed to a high or low resistive state according to the truth values of the input vector. To evaluate the Boolean function, the paradigm relies on the presence (absence) of a path from input to output to determine whether the function evaluates to true (false).

Noteworthy research efforts have been dedicated to developing design flows for synthesizing compact crossbar designs [16], [17]. The design flows consist of a complex hierarchy of small, intermediate synthesis steps. These steps involve mapping the functional representation between different data structures (e.g. NNFs [19], AIGs [20], BDDs [21]) and optimizing the size of the representations. Equivalence checking is used to verify the functional correctness of each of the intermediate steps. Formally, equivalence checking is the problem of verifying the equivalence of two different circuit representations. While effective synthesis techniques have successfully been developed, the capabilities of equivalence checking techniques for flow-based computing are lagging behind. For example, the COMPACT synthesis tool can synthesize functions into crossbars with dimensions up to 512×512 in seconds [17]. However, it takes hours for existing equivalence checking techniques to verify the functional correctness of the designs [16], [17], [22]. The development of scaleable verification techniques will open the door to further advancing the frontiers of flow-based in-memory computing.

Traditional VLSI circuits are represented using *directed acyclic graphs* (DAGs). SAT solving has been proven to be a successful technique for combinational equivalence checking (CEC) for such circuits. In CEC, the outputs for two circuits are compared and verified to be the same for all input assignments using a *miter circuit* (see more details in Section II-B). One circuit is the so-called golden model (specification) and the other is the circuit to be verified [23]. Then, the miter

circuit is converted into conjunctive normal form (CNF) using Tseitin's transformation [24]. Finally, the CNF formula is fed to a SAT solver, such as MiniSAT [25]. The equivalence, or non-equivalence of the two circuits is determined by the solution to the SAT formulation. The outlined approach is capable of verifying traditional CMOS circuits with tens of millions of combinational gates.

Unfortunately, it is not straightforward to directly apply the SAT-based approach to verify crossbars designs for flow-based computing. This is due to the undirected nature of the underlying computational graph. Cycles in the graph effectively break the SAT formulation, which we show using a detailed example (see Section III). While techniques to handle directed graphs with some cycles have been investigated [26]–[28], none of the techniques can directly be applied to undirected computational graphs. Existing studies on equivalence checking for flow-based computing are based on brute-force enumeration [16], graph reachability [29], and recurrent neural network (RNN) formulations [22].

Verification achieved by the means of brute-force enumeration is unsurprisingly not very scalable. In [22], a crossbar design was modeled as a RNN. This allows the exploitation of massive parallel computations of GPUs to speed up the brute-force enumeration. Subsequent work [29] relies on graph reachability, which was demonstrated to perform better for most circuits. However, none of the proposed techniques can verify circuits of even moderate size. A window for opportunity exists as these techniques do not exploit the full potential of established verification techniques, which rely on decades of advances within SAT solving [30]–[32]

In this paper, we propose a framework called XSAT for equivalence checking of crossbar designs for flow-based computing. The XSAT framework draws on advances within bounded model checking (BMC), which is classically used for property checking. The main idea is to use BMC to unroll the circuit and discretize time, which effectively converts the undirected graph into a directed acyclic graph. Next, equivalence checking can be performed using traditional SAT formulations. The limitation of the approach is the unrolling, which substantially increases the size of the circuit. To handle the larger problem size, XSAT uses a divide-and-conquer approach to decompose the large problem into multiple smaller subproblems that can be solved in parallel.

In this paper, we make the following contributions:

1) We propose to use bounded model checking (BMC) to verify the correctness of a crossbar design using traditional SAT solving techniques.
2) To improve scalability, we propose a divide-and-conquer technique that exploits parallelism. By fixating input variables, the problem can be split into two smaller subproblems, which can be solved in parallel.
3) Our experimental evaluation on nineteen MCNC circuits shows that our proposed framework is capable of solving 19/19 benchmarks, whereas the previous state-of-the-art equivalence checking technique is only capable of solving 11/19 benchmarks.

The remainder of the paper is organized as follows: preliminaries are provided in Section II. The problem is defined in Section III, and we give an overview of XSAT in Section IV. We conduct experimental results in Section V and conclude in Section VI.

II. Preliminaries

A. Flow-based computing

Flow-based computing is a digital in-memory computing paradigm that leverages memristor crossbar arrays to perform in-memory computations. A memristor crossbar array is a two-dimensional circuit of dimensions $M \times N$, consisting of wordlines, bitlines, and memristors. The wordlines (rows) $R_i, 0 \leq i < M$, are horizontal nanowires, and the bitlines (columns) $C_j, 0 \leq j < N$, are vertical nanowires. At each intersection of wordlines R_j and bitline C_j is a memristor m_{ij}, connecting both wordline and bitline. A memristor is a non-volatile resistive device with either a high or low resistive state.

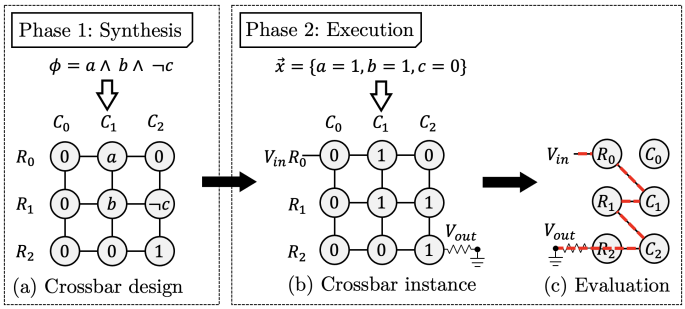


Fig. 1. Overview of flow-based computing.

The first phase, the synthesis phase, produces a crossbar design $\mathcal{D}$ for a Boolean function ϕ. The input of the synthesis phase is a specification of ϕ in a hardware descriptive language such as Verilog. In Figure 1(a), an example of a specification for the Boolean function $\phi = a \wedge b \wedge \neg c$ is provided. The output of the synthesis phase is a crossbar design $\mathcal{D}$. Let $\{x_0, ..., x_n\}$ be the Boolean variables of ϕ, then a crossbar design $\mathcal{D}$ is an assignment of these Boolean variables $\{x_0, ..., x_n\}$, the negation of these Boolean variables $\{\neg x_0, ..., \neg x_n\}$, and the truth values 0 and 1 to the memristors. In Figure 1(a), the resulting crossbar design $\mathcal{D}$ is shown for the Boolean function ϕ.

The second phase is the execution phase. In this phase, the crossbar design is leveraged for computation. During execution, an input vector is provided where the Boolean variables are assigned a truth value. Based on these truth values, their corresponding memristors are programmed to the respective resistive states. This is illustrated in Figure 1(b) where the input vector $\vec{x} = \{a = 1, b = 1, c = 0\}$ is provided. The memristors are programmed accordingly. Finally, a high input voltage is applied to the input wordline and the output wordline is grounded. For this step, the crossbar array is modeled as a bipartite graph $G = (V, E)$ where V is a set of nodes and E is a set of edges, as shown in Figure 1(c). Observe that the nodes V correspond to the nanowires R_i and C_j. The edges correspond to the memristors m_{ij}. There exists an edge between two nodes R_i and C_j if and only if

the memristor m_{ij} is in a low resistive state (1). Otherwise, when the memristor m_{ij} is in a high resistive state (0), there is no edge between R_i and C_j. When a high input voltage is applied to the input wordline, this will result in an electrical current flowing through the crossbar from the input wordline to the output wordline. When an electrical current is present on a wordline R_i (bitline C_i), and a connected memristor m_{ij} is in a low resistive state, the electrical current will dissipate onto the connected bitline C_j (wordline R_j). When the electrical current reaches the output wordline, the Boolean function ϕ evaluates to true. Otherwise, when no electrical current can be sensed, the Boolean function ϕ evaluates to false. In Figure 1(c), we observe that there is a path in the bipartite graph from the input wordline to the output wordline, and thus the Boolean function ϕ evaluates to true.

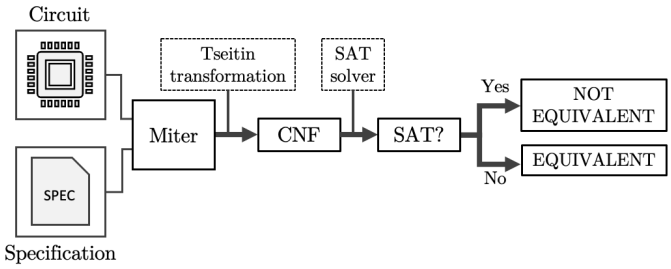


Fig. 2. Overview of equivalence checking using SAT solving.

B. Formal verification

Formal verification is categorized into two broad classes: model checking and equivalence checking [33]. While the former tends to concern itself with regard to properties, the latter concerns itself with verifying whether specification and final design are functionally equivalent. However, solving techniques from one class may be used to alleviate solving a problem from the other class. We will first provide a short review of both SAT solving and bounded model checking.

1) SAT solving: One of the most well-known approaches for equivalence checking is by transforming the problem into a SAT problem. In a SAT problem, we are provided a Boolean function in conjunctive normal form (CNF), and we ask the solver whether there is at least one input vector for which the Boolean function evaluates to true. When one such input vector can be found, the problem is SAT, which entails that the circuit and the specification are not equivalent. Otherwise, when no such input vector can be found, the problem is UNSAT and both circuit and specification are equivalent. This existential problem is NP-Complete [34]. To transform the problem into SAT, a miter is constructed of the specification and the circuit. A miter is an exclusive or (XOR) operation on the output of both the specification and the circuit. In case of multiple outputs, a XOR operation is applied to the respective outputs, which are all captured by an or (OR) operation. A miter construction of the specification and circuit is illustrated in Figure 2. Traditionally, the next step is to convert the Boolean function into conjunctive normal form (CNF) using Tseitin's transformation. However, some intermediary data structures, such as And-Inverter Graphs (AIGs), may be employed [23]. It has been shown that doing so can result in great improvements.

2) Bounded model checking: A complementary approach to SAT solving for equivalence checking is model checking. In model checking, the circuit is transformed into a finite state machine where the transitions between states are defined using temporal logic [35]. In essence, the circuit is unrolled for a number of time steps. Bounded model checking is a subset of model checking where an upper bound is placed on the number of time steps. An important concept is the completeness threshold. When a property does not violate the model up until this completeness threshold, then the model and the property are equivalent. Lastly, a temporal model for bounded model checking can be reduced into SAT. We will use this reduction to verify the correctness of a crossbar design. More specifically, we will first translate the crossbar design into a finite state machine with an upper bound on the number of time steps, and then show equivalence/non-equivalence between the model and the specification by solving a SAT formulation.

III. Problem Formulation and Challenges

The problem this paper addresses is defined as follows: given a crossbar design $\mathcal{D}$ and a specification ϕ, verify if $\mathcal{D}$ and ϕ are equivalent:

$$\mathcal{D} \stackrel{?}{\equiv} \phi$$

The objective is to minimize the overall runtime of the verification.

We explain why the standard combinational equivalence checking using SAT formulations cannot be applied in Figure 3. An undirected crossbar is shown in Figure 3(a). The logic expressions for the rows and columns are shown in Figure 3(b). R_i and C_j denote row i and column j in the crossbar, respectively. It can be observed that R_0 is a function of R_0 in Figure 3(c). This results in that even if no input voltage is applied, R_0 can potentially drive R_0 due to the cyclic dependency. This cyclic dependency breaks the correctness of CNF-based SAT formulations using miter circuits.

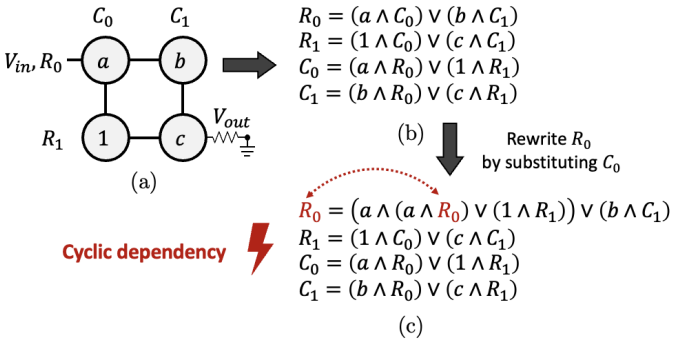


Fig. 3. Example of cyclic dependency.

Several investigations have been performed into CMOS circuits with directed computational graphs with cycles [26]–[28]. However, those techniques cannot be used for undirected computational graphs. This promoted the research explorations into equivalence checking using brute-force enumeration and graph reachability [16], [22], [29].

IV. Proposed XSAT framework

In this section, we introduce the proposed XSAT framework. An overview of the framework is illustrated in Figure 4.

The input of the framework are the crossbar design $\mathcal{D}$ and the specification ϕ. The output of the framework is whether the both $\mathcal{D}$ and ϕ are equivalent or non-equivalent. The framework consists of two components: (1) a base approach using BMC, and (2) an extension using a divide-and-conquer technique. In Section IV-A, we will first explain the base approach where we formulate a bounded checking formulation for the whole crossbar through unrolling of time steps. Then, in Section IV-B, we introduce the divide-and-conquer approach where we divide the problem into smaller subproblems by fixating input variables.

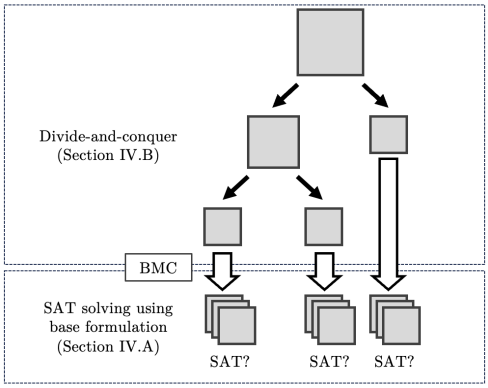


Fig. 4. High-level overview of the XSAT framework.

A. Base formulation

In this section, we translate the crossbar design into a formulation for bounded model checking. The input is a crossbar design $\mathcal{D}$ and the specification ϕ. The output is equivalent/non-equivalent. The framework consists of three steps: graph conversion, graph compression, and bounded model checking. An overview of the flow is provided in Figure 5. Further, we illustrate our proposed bounded model checking approach with an example in Figure 6.

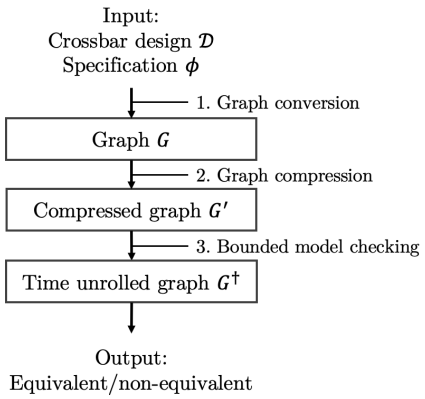


Fig. 5. Overview of bounded model checking.

1) Graph conversion: The first step is to convert the crossbar design $\mathcal{D}$ into an undirected bipartite graph G. For each row R_i and column C_j, we introduce a node R_i and C_j, respectively. In Figure 6(a), we show a small crossbar design with three rows and two columns. Consequently, its resulting undirected graph $G = (V, E)$ has nodes $V = \{R_0, R_1, R_2, C_0, C_1\}$ and edges $E = \{(R_0, C_0, a), (R_0, C_1, b), ...(R_2, C_1, d)\}$). Each edge is a triple, where its first two elements are the nodes, and the third element is the literal. The resulting graph is shown in Figure 6(b).

2) Graph compression: The next step is graph compression, which consists of two sub-steps: 1) edge removal and 2) node contraction. In the edge removal step, we remove edges with label '0'. When an edge has label '0', the corresponding memristor will be programmed to a high resistive state (OFF). This entails that no electrical current will ever be able to flow through the device. Hence, we can remove the edge from the graph without changing its behaviour. In Figure 6(c), we observe how the edge between node R_2 and C_0 has been removed. In the node contraction step, we contract two nodes connected by an edge with label '1'. When an edge has label '1', the corresponding memristor is in a low resistive state (ON). When an electrical current is present on a wordline (bitline), the electrical current will always flow through this device onto its connecting bitline (wordline). Consequently, two nodes R_i and C_j connected by an edge $(R_i, C_j, 1)$ are deemed *equivalent*, and we can contract the two nodes R_i and C_j. These observations were also made in [29]. In Figure 6(d), we observe how node R_1 has merged into node C_0 such that only node C_0 remains.

3) Bounded model checking: In this last step, the compressed graph G'' is converted into a model for SAT solving based on bounded model checking. Let $G'' = (V, E)$ be the undirected graph where V is a set of nodes representing the nanowires (wordlines and bitlines), and E is a set of edges representing the memristors. As discussed in Section II-B2, bounded model checking consists of time unrolling. For each state, we will introduce time steps. The number of time steps has both an upper bound and a lower bound. The lower bound is one, and the upper bound is the number of nodes in the graph. The number of nodes is the upper bound because at most $|V|$ unique nodes can be visited. Define $T = |V|$ as the maximum number of time steps and let M be the number of rows, and N the number of columns. Then we model the behaviour of the crossbar design as follows:

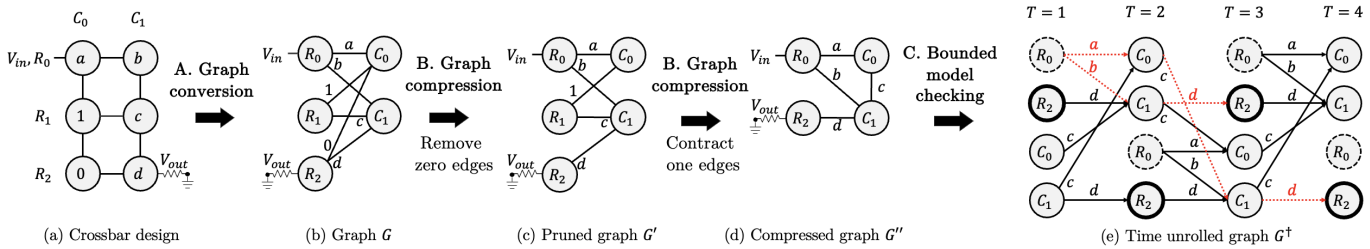
$$R_1^t = 1, \qquad t \in [1, T] \quad (1)$$

$$R_i^t = \bigvee m_{ij} \wedge C_j^{t-1}, \quad \forall i \in [2, M], j \in [1, N], t \in [2, T] \quad (2)$$

$$C_j^t = \bigvee m_{ij} \wedge R_i^{t-1}, \quad \forall i \in [2, M], j \in [1, N], t \in [2, T] \quad (3)$$

$$\phi = \bigvee R_M^t, \qquad \forall t \in [1, T] \quad (4)$$

The equation on line 1 defines that the input nanowire R_1 must be true (ON) for all time steps. This is because the input voltage will be high during the evaluation, thus providing an electrical current on the input nanowire. The equation on line 2 defines how row R_i can be reached in time step t. As discussed in Section III, a row R_i can only be true (ON) at time step t if and only if a column C_j is true (ON) at time step $t-1$ and the



(a) Crossbar design (b) Graph G (c) Pruned graph G' (d) Compressed graph G'' (e) Time unrolled graph $G^\dagger$

Fig. 6. Example of bounded model checking.

memristor at their intersection m_{ij} is true (ON). In physical terms, this means that an electrical current can only reach row R_i if and only if there is an electrical current present on column C_j and the memristor at their intersection is in a low resistive state. Similarly for a column C_j on line 3. Finally, we define on line 4 the Boolean function of the crossbar design $\mathcal{D}$ as the collection of all possible paths from the input nanowire to the output nanowire. More specifically, we capture these paths as a disjunction of all possible paths for different time steps in the range from 1 to the upper bound T.

In Figure 6(e), we illustrate this model graphically. The input nanowire R_0 is highlighted with a dashed line, and the output nanowire is highlighted with a bold line. The graph is unrolled for $T = |V| = 4$ time steps. Further, the edges represent the relations between time step t and $t + 1$. For example, column C_0 in time step $T = 2$ can be reached from row R_0 in time step $T = 1$ using the memristor a or from column C_1 in time step $T = 1$ using the memristor c. We have highlighted the paths from input nanowire to output nanowire using red dashed lines. For example, there is a path $R_0^1 \xrightarrow{a} C_0^2 \xrightarrow{c} C_1^3 \xrightarrow{d} R_2^4$. This path denotes $\phi = a \wedge c \wedge d$. The other path is $R_0^1 \xrightarrow{b} C_1^2 \xrightarrow{d} R_2^3$.

B. Divide-and-conquer

In this section, we introduce a divide-and-conquer technique to leverage multi-core parallelism on computing machines. The idea is to fixate a set of variables X to reduce the problem size. When fixating a variable x, we obtain two independent subproblems, one for $x = 0$ and one for $x = 1$. Each individual problem has smaller or equal problem size as the original problem. The divide-and-conquer technique consists of three steps: variable sorting and variable fixating, and graph compression. The last step is the same as the graph compression step explained in Section IV-A. Hence, we will omit the details here. The divide-and-conquer technique can best be described as the construction of a binary tree. The root node of the tree is the original problem. Each internal node has two outgoing branches, one for $x = 0$ and one for $x = 1$. The leaf nodes describe the subproblems that will be solved in parallel. The depth of each path from root node to leaf node is defined by two user-defined parameters: the maximum depth D, and a threshold T. The first parameter, D defines how many subproblems can be solved maximally in parallel (2^D). The second parameter, T, defines a maximum problem size. The problem size is defined by the number of nodes $|V|$ in the graph $G = (V, E)$. When a subproblem is smaller than the threshold, the problem can be solved quickly by the SAT solver. Hence, there is no need to further reduce the problem. In Algorithm 1, we describe the divide-and-conquer technique. The input of the algorithm is the graph G, constructed from the crossbar design $\mathcal{D}$, and the user-defined parameters D and T. The output of the algorithm is a set of subgraphs $\mathcal{G}$. Now, we will elaborate on the main steps of the divide-and-conquer algorithm.

1) Variable sorting: In this first step, we count the occurrences for each unique variable in the crossbar design. Next, these variables are sorted in descending order such that we obtain an ordered list X. For example, in Figure 7 for $D = 0$, we observe that variable a occurs 2 times, variable b occurs 3 times, and variables c and d one time. The corresponding ordered list $X = \{b, a, c, d\}$.

2) Fixating variable: In this step, we construct a set of small subgraphs based on the original graph G. These subgraphs are constructed in an iterative manner. We introduce an auxiliary variable $\mathcal{H}$, which holds the subgraphs. Initially, $\mathcal{H}$ only holds G (line 3). In each iteration, we will fixate a variable x. If the number of iterations i exceeds the maximum depth D or the graph G is smaller than the threshold T, then we add the subgraph to the output set $\mathcal{G}$ (lines 7-10). Otherwise, we fixate the variable x for $x = 0$ and $x = 1$, and we add these subproblems to the auxiliary set $\mathcal{H}$ for the next iteration (lines 11-13). Finally, we increment the counter i (line 14).

Algorithm 1 Divide-and-conquer algorithm

Input: G, T, D
Output: $\mathcal{G}$ // A set of subproblems
1: **function** DIVIDEANDCONQUER(G)
2: $X \leftarrow$SORTEDOCCURRENCES(G)
3: $\mathcal{H} \leftarrow \{G\}$, $\mathcal{G} \leftarrow \emptyset$, $i \leftarrow 0$
4: **while** $\mathcal{H} \neq \emptyset$ **do**
5: $x \leftarrow X[i]$
6: $G \leftarrow$ POP($\mathcal{H}$)
7: **if** $i \geq D \vee |G| \leq T$ **then**
8: $\mathcal{G} \leftarrow \mathcal{G} \cup \{G\}$
9: **continue**
10: **end if**
11: $G^- \leftarrow$ FIXATEANDCOMPRESS(G, x, 0)
12: $G^+ \leftarrow$ FIXATEANDCOMPRESS(G, x, 1)
13: $\mathcal{H} \leftarrow \mathcal{H} \cup \{G^-, G^+\}$
14: $i \leftarrow i + 1$
15: **end while**
16: **return** $\mathcal{G}$
17: **end function**

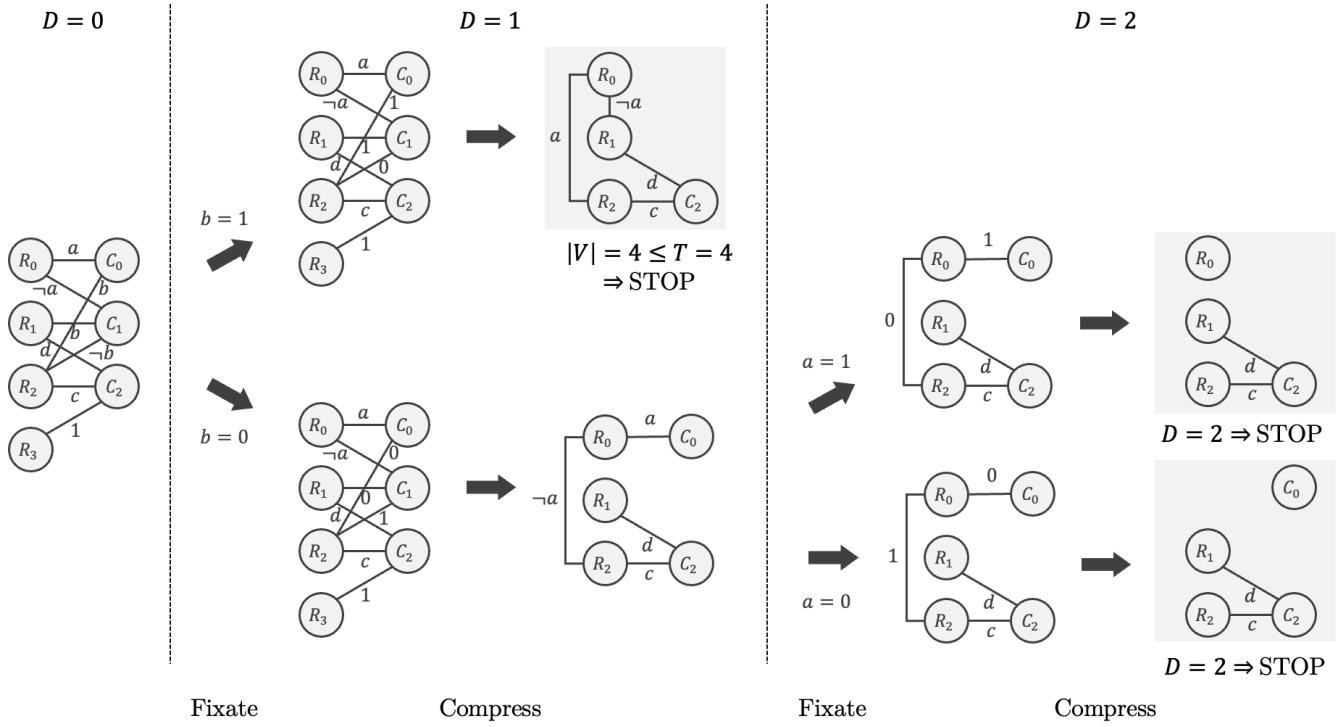


Fig. 7. Example of the proposed divide-and-conquer algorithm. The user-defined parameter are $T = 4$ and $D = 2$.

In Figure 7, we illustrate the construction of the subgraphs based on Algorithm 1. The input is the graph in column $D = 0$, and the user-defined parameters are $T = 4$ and $D = 2$. In Figure 7, in column $D = 1$, we observe that first variable b is fixated. This results in two subgraphs, one for $b = 0$ and one for $b = 1$. For $b = 1$, we observe that the number of nodes of the graph after compression is smaller than the threshold T ($|V| = 4 \leq T$). Hence, this subgraph branches no further. For $b = 0$, the threshold is not yet met, and in subsequent iteration, variable a is fixated. Again, this results in two subgraphs, one for $a = 1$ and one for $a = 0$. We have reached the maximum depth $D = 2$, and the algorithm halts. The output is the set of the three subgraphs in the gray boxes.

V. Experimental evaluation

In this section, we evaluate our proposed equivalence checking technique for flow-based computing. Experimental results are conducted on a machine with 20 Intel Core i9-9900X processors and a memory of 128GB. Python 3.8 was used for the code implementation in combination with the ABC tool [33] for the combinational equivalence checking based on MiniSAT. The source code is publicly available on GitHub[1]. We evaluate our proposed equivalence checking technique XSAT on 26 MCNC benchmarks [36]. In Table I, we provide the properties for these benchmarks and their respective crossbar designs. These benchmark properties are the number of input variables, and the number of output variables. The crossbar properties are the number of rows and columns, the number of literals, and the number of memristors that are programmed ON and OFF. The benchmarks are subdivided into two categories: nineteen smaller benchmarks (rows plus columns are less than 800), and seven large benchmarks.

[1]https://github.com/sventhijssen/xsat

TABLE I
Overview of 26 benchmarks from the MCNC benchmark suite [36].

Benchmark	Properties		Crossbar				
	Inputs (num)	Outputs (num)	Rows (num)	Cols (num)	Lits (num)	ON (num)	OFF (num)
SMALL							
frg1	28	3	59	74	181	22	4163
e64	65	65	112	76	157	30	8325
count	35	16	65	41	159	9	2497
cht	47	36	45	55	146	2	2327
b9	41	21	96	93	266	28	8634
bcd	26	38	280	247	575	29	68556
i6	138	67	83	143	426	7	11436
term1	34	10	121	125	365	31	14729
bca	26	46	271	228	548	23	61217
i7	199	67	201	169	663	4	33302
example2	85	66	208	190	570	29	38921
bc0	26	11	562	571	1845	120	318937
i3	132	6	71	95	198	33	6514
i5	133	66	145	133	410	26	18849
comp	32	3	71	93	242	23	6338
my_adder	33	17	102	115	361	17	11352
x4	94	71	245	214	645	56	51729
too_large	38	3	250	257	780	74	63396
x1	51	35	364	349	1099	100	125837
LARGE							
spla	16	46	451	391	1185	56	175100
misex3	14	14	533	523	1599	94	277066
alu4	14	8	478	549	1765	82	260575
dalu	75	16	584	572	1727	60	332261
apex5	117	88	664	585	2108	98	386234
apex1	45	45	717	702	2143	96	501095
seq	41	35	711	713	2095	88	504760

TABLE II
RUNTIMES FOR EQUIVALENCE CHECKING TECHNIQUES CHECK, RNN, AND PROPOSED XSAT BASE FORMULATION ON 19 MCNC BENCHMARKS. VERIFICATION FOR BOTH EQUIVALENT DESIGNS AND NON-EQUIVALENT DESIGNS.

Benchmark	Equivalent							Non-Equivalent						
	CHECK [29]	RNN [22]	XSAT Base					CHECK [29]	RNN [22]	XSAT Base				
	Total (s)	Total (s)	Time steps (num)	Pre (s)	Write (s)	CEC (s)	Total (s)	Total (s)	Total (s)	Time steps (num)	Pre (s)	Write (s)	CEC (s)	Total (s)
frg1	8.1	672.6	111	0.1	0.2	4.9	5.1	-	369.0	110	0.1	0.2	5.0	5.1
e64	22.3	-	158	0.2	0.2	1.1	1.3	0.1	-	158	0.2	0.2	0.2	0.4
count	54.4	-	97	0.1	0.1	0.4	0.5	28.8	-	97	0.1	0.1	0.2	0.3
cht	62.9	-	98	0.1	0.2	4.9	5.1	15.9	-	97	0.1	0.2	1.3	1.5
b9	416.4	-	161	0.1	0.3	10.8	11.2	81.3	-	161	0.1	0.4	7.4	7.8
bcd	559.7	5.0	498	1.7	2.6	37.6	40.2	163.1	0.9	582	2.3	4.7	84.2	88.9
i6	1073.9	-	219	0.2	1.2	24.0	25.2	54.1	-	218	0.2	1.3	2.1	3.3
term1	1225.2	-	215	0.3	0.6	25.2	25.9	518.6	-	215	0.3	0.7	2.2	2.9
bca	2742.6	16.1	476	1.4	2.4	52.9	55.2	121.9	2.1	916	5.6	11.8	31.2	43.0
i7	2866.5	-	366	0.7	6.1	115.8	121.9	14.8	-	365	0.7	6.1	61.7	67.8
example2	3199.8	-	369	0.8	1.9	142.4	144.3	553.3	-	368	0.8	2.0	73.4	75.4
bc0	-	228.6	603	2.8	6.2	539.5	545.7	-	25.0	603	2.8	6.0	110.6	116.7
i3	-	-	133	0.1	0.2	5.0	5.2	-	-	132	0.1	0.2	3.2	3.5
i5	-	-	252	0.4	0.8	18.4	19.2	583.8	-	252	0.4	0.8	13.7	14.5
comp	-	-	141	0.1	0.3	15.1	15.4	-	-	140	0.1	0.3	10.4	10.7
my_adder	-	-	200	0.2	0.6	23.0	23.7	-	-	200	0.2	0.6	3.8	4.4
x4	-	-	403	1.1	2.3	72.1	74.4	376.0	-	402	1.2	2.5	52.6	55.0
too_large	-	-	433	1.5	3.3	322.0	325.3	-	-	433	1.6	3.5	49.4	52.9
x1	-	-	613	3.0	6.1	348.6	354.7	-	-	612	3.2	6.7	184.7	191.4
Completed	11/19	4/19					**19/19**	12/19	4/19					**19/19**

First, in Section V-A, we will compare our proposed XSAT framework with other state-of-the-art equivalence checking techniques for flow-based computing. Then, in Section V-B, we will evaluate our proposed divide-and-conquer technique.

A. Comparison with previous techniques

In this section, we will compare our proposed framework XSAT with previous work. In Table II, we provide the runtime for equivalence checking of both equivalent crossbar designs and non-equivalent crossbar designs. To create the non-equivalent crossbar designs, we introduce a single error by programming a random literal in the crossbar design to ON/OFF. We compare our proposed XSAT framework with CHECK, a technique based on graph reachability [29], and with the technique based on RNNs [22]. For our proposed framework, we provide a detailed runtime analysis of the different steps. These steps are the pre-processing (pruning), the file creation to write the bounded model checking, and the combinational equivalence checking (CEC). The total runtime is the sum of the runtimes for these substeps. The maximum runtime for the equivalence checking techniques is one hour (3600s). Runtimes that exceed this limit are indicated by a dash ($-$).

First, we analyze the equivalent case. From the results, we observe that CHECK is capable of solving $11/19$ benchmarks whereas the RNN framework is capable of solving $4/19$ benchmarks. We observe that in many cases, the RNN framework does not scale well to benchmarks with a large number of input variables. For example, the technique can handle up to 28 input variables. CHECK on the other hand, cannot handle benchmarks with a large number of literals and memristors programmed ON. This is due to that the number of paths in the graph grows in this number of literals. For example, benchmarks *x1* and *bc0* have 1099 and 1845 literals, respectively. Finally, we observe that XSAT is capable of solving all benchmarks. We can also observe that the runtime for XSAT is much smaller than the runtime for both CHECK and the RNN-based technique for the majority of benchmarks. Thus, we conclude that XSAT outperforms both CHECK and the RNN-based technique to show equivalence.

Now, we analyze these techniques for the non-equivalent case. We observe that CHECK is capable of solving $12/19$ benchmarks, which is one more than the number of benchmarks for equivalence checking. This is because the complexity of the problem may reduce. We also observe that the runtime decreases for the majority of benchmarks. However, for some benchmarks, the problem becomes more difficult, such that CHECK is no longer capable of solving the problem within one hour. An example is the benchmark *frg1*. The RNN-based technique is still capable of solving 4 out of 19 benchmarks. As long as the number of input variables remains unchanged, the RNN-based technique will have similar runtime to its equivalent counterpart. Finally, we observe that our proposed XSAT framework is again capable of solving all 19 benchmarks. The runtimes to illustrate non-equivalence are more or less similar to the runtimes for equivalence.

B. Evaluation of divide-and-conquer technique

In this section, we will evaluate the proposed divide-and-conquer technique. First, we will determine good values for the user-defined parameters D and T. Recall that the parameter D is the maximum depth of the binary tree, and T is the maximum number of nodes in the graph.

In Figure 8, we plot the runtime for XSAT in terms of the number of time steps for the small benchmarks. The number of time steps is equal to the number of rows plus the number of columns minus the number of memristors which are programmed ON. The latter subtraction is due to the graph contraction. From the figure, we observe that equivalence can be shown within several minutes when the number of time

TABLE III

COMPARISON OF THE BASE FORMULATION AND THE DIVIDE-AND-CONQUER TECHNIQUE ON 7 MCNC BENCHMARKS. FOR BOTH TECHNIQUES, THE RUNTIME FOR THE PRE-PROCESSING, THE TIME TO WRITE THE FILE, AND THE RUNTIME FOR COMBINATIONAL EQUIVALENCE CHECKING ARE PROVIDED. PROPERTIES FOR THE SAT PROBLEMS, I.E., THE NUMBER OF CLAUSES AND LITERALS, ARE ALSO PROVIDED.

Benchmark	XSAT base formulation						XSAT using divide-and-conquer technique					
	Pre (s)	Write (s)	CEC (s)	Total (s)	Clauses (num)	Literals (num)	Pre (s)	Write (s)	CEC (s)	Total (s)	Clauses (num)	Literals (num)
spla	3.90	9.65	2232.31	2245.86	3197427	6408444	35.16	0.39	0.50	36.06	59741	161408
misex3	6.64	14.78	3294.74	3316.16	5210060	10443593	12.18	0.45	0.80	13.44	179515	501609
alu4	6.28	16.67	3516.00	3538.95	5296423	10622106	16.12	0.40	0.83	17.35	164110	450064
dalu	7.99	19.16	3448.04	3475.19	6083041	12188311	65.78	2.02	58.87	126.67	222249	446825
apex5	10.51	26.70	1621.72	1658.92	*	*	133.72	2.58	156.41	292.71	702476	1421614
apex1	13.60	30.99	10083.48	10128.07	*	*	70.43	1.67	157.52	229.62	278460	559333
seq	13.21	28.73	7208.54	7250.48	*	*	90.65	2.25	17.38	110.28	*	*
Normalized average	**1.00**	1.00	1.00	1.00	1.00	1.00	6.63	**0.06**	**0.02**	**0.04**	**0.03**	**0.04**

steps is below 400. When the number of time steps is above the threshold of 400, the runtime starts an exponential growth. Further, from the benchmarks in Table II, we conclude that when the number of time steps is lower than 200, the runtime is less than a minute. Hence, for the proposed divide-and-conquer technique, we will set the user-defined parameter $T = 200$. We also observe in Figure 8 that the number of SAT clauses grows large as the number of time steps grows. In the figure, we can observe that the runtime strongly correlates with the number of SAT clauses.

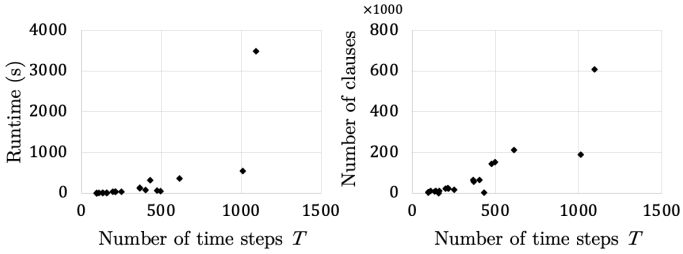


Fig. 8. Runtime in seconds and number of clauses for different number of time steps.

Next, we will analyze the effects of the user-defined parameter D on the runtime of the XSAT algorithm. In Figure 9, we plot the maximum runtime for the SAT solver. We observe that the runtime decreases as the number of subproblems increases. This is due to the subproblems being smaller when fixating a variable. Finally, we conclude that for $D = 10$, the runtime barely improves. In Figure 9 on the right, we show the distribution of the number of occurrences for the variables in decreasing order. The highest count for a variable is 228 and the smallest count is 1. We observe that the variable occurrence drops sharply until it reaches a sharp inflexion point for the eleventh most occurring variable and then stagnates near zero. This point coincides with the point where there is no further improvement from our divide-and-conquer technique. Based on this sweep and analysis, we set the user-defined parameter for the maximum depth to $D = 10$ for our last experiment.

Finally, we will compare our base formulation with the divide-and-conquer approach. In Table III, we provide the runtime for our base formulation and our dynamic approach using the divide-and-conquer technique for seven of the largest benchmarks. For some benchmarks, no clauses and literals are

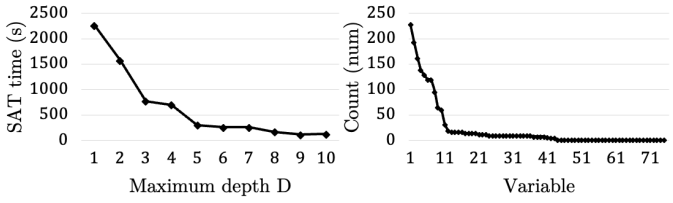


Fig. 9. Runtime in terms of the maximum depth D for the divide-and-conquer technique, and distribution of the number of occurrences for variables in the crossbar design for the benchmark *dalu*.

reported, indicated by an asterisk (*), as the pre-processing in ABC shows equivalence before feeding the circuit to the SAT solver. We will primarly look at the total runtime, which is much smaller for the divide-and-conquer technique on average over all benchmarks. Using the divide-and-conquer technique, the runtime improves by 93× compared with the base formulation. The 93× is computed as the normalized average over the improvements of the individual benchmarks.

VI. CONCLUSION AND FUTURE WORK

Flow-based computing is a promising digital in-memory computing paradigm. A lot of effort has been made on synthesis for the computing paradigm, yet a scalable verification technique was non-existent. We have illustrated that equivalence checking for flow-based computing is a challenging problem due to the nature of memristor crossbars. Namely, the bidirectional behavior of memristors challenge existing and established equivalence checking techniques. The model for flow-based computing is an undirected graph, which is fundamentally different from directed (acyclic) graphs in traditional CMOS-based circuits. In this paper, we have proposed a new equivalence checking technique for flow-based computing based on bounded model checking. Further, we have introduced a divide-and-conquer technique to reduce the overall problem into smaller subproblems, which can be solved in parallel. This results in an average speed-up of 93× over seven larger MCNC benchmarks. We have compared our work with the previous state-of-the-art equivalence checking techniques for flow-based computing on 19 MCNC benchmarks, and conclude that our proposed technique is the only one which can solve all benchmarks within one hour.

REFERENCES

[1] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Information sciences*, vol. 275, pp. 314–347, 2014.

[2] J. Fan, F. Han, and H. Liu, "Challenges of big data analysis," *National science review*, vol. 1, no. 2, pp. 293–314, 2014.

[3] J. Qiu, Q. Wu, G. Ding, Y. Xu, and S. Feng, "A survey of machine learning for big data processing," *EURASIP Journal on Advances in Signal Processing*, vol. 2016, pp. 1–16, 2016.

[4] L. Zhou, S. Pan, J. Wang, and A. V. Vasilakos, "Machine learning on big data: Opportunities and challenges," *Neurocomputing*, vol. 237, pp. 350–361, 2017.

[5] Y. Li, C. Huang, L. Ding, Z. Li, Y. Pan, and X. Gao, "Deep learning in bioinformatics: Introduction, application, and perspective in the big data era," *Methods*, vol. 166, pp. 4–21, 2019.

[6] F. Emmert-Streib, Z. Yang, H. Feng, S. Tripathi, and M. Dehmer, "An introductory review of deep learning for prediction models with big data," *Frontiers in Artificial Intelligence*, vol. 3, p. 4, 2020.

[7] J. Backus, "Can programming be liberated from the von neumann style? a functional style and its algebra of programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[8] N. Verma, H. Jia, H. Valavi, Y. Tang, M. Ozatay, L.-Y. Chen, B. Zhang, and P. Deaville, "In-memory computing: Advances and prospects," *IEEE Solid-State Circuits Magazine*, vol. 11, no. 3, pp. 43–55, 2019.

[9] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," *Nature nanotechnology*, vol. 15, no. 7, pp. 529–544, 2020.

[10] M. Hu, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J. J. Yang, and R. S. Williams, "Dot-product engine for neuromorphic computing: Programming 1t1m crossbar to accelerate matrix-vector multiplication," in *Proceedings of the 53rd annual design automation conference*, pp. 1–6, 2016.

[11] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Magic—memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[12] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-based material implication (imply) logic: Design principles and methodologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 10, pp. 2054–2066, 2013.

[13] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proceedings of the 53rd Annual Design Automation Conference*, pp. 1–6, 2016.

[14] A. Velasquez and S. K. Jha, "Parallel boolean matrix multiplication in linear time using rectifying memristors," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1874–1877, IEEE, 2016.

[15] A. U. Hassen, D. Chakraborty, and S. K. Jha, "Free binary decision diagram-based synthesis of compact crossbars for in-memory computing," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 5, pp. 622–626, 2018.

[16] D. Chakraborty and S. K. Jha, "Design of compact memristive in-memory computing systems using model counting," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4, 2017.

[17] S. Thijssen, S. K. Jha, and R. Ewetz, "Compact: Flow-based computing on nanoscale crossbars with minimal semiperimeter," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 232–237, IEEE, 2021.

[18] M. R. H. Rashed, S. Thijssen, S. K. Jha, F. Yao, and R. Ewetz, "Stream: Towards read-based in-memory computing for streaming based processing for data-intensive applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[19] S. K. Jha, D. E. Rodriguez, J. E. Van Nostrand, and A. Velasquez, "Computation of boolean formulas using sneak paths in crossbar computing," Apr. 19 2016. US Patent 9,319,047.

[20] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Fast logic synthesis for rram-based in-memory computing using majority-inverter graphs," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 948–953, IEEE, 2016.

[21] D. Chakraborty and S. K. Jha, "Automated synthesis of compact crossbars for sneak-path based in-memory computing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 770–775, IEEE, 2017.

[22] S. Singireddy, R. Ewetz, and S. Jha, "Deep learning toolkit-driven equivalence checking of flow-based computing systems," in *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pp. 50–53, IEEE, 2022.

[23] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pp. 836–843, 2006.

[24] G. S. Tseitin, "On the complexity of derivation in propositional calculus," *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pp. 466–483, 1983.

[25] N. Sorensson and N. Een, "Minisat v1. 13-a sat solver with conflict-clause minimization," *SAT*, vol. 2005, no. 53, pp. 1–2, 2005.

[26] S. Malik, "Analysis of cyclic combinational circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 7, pp. 950–956, 1994.

[27] M. Riedel and J. Bruck, "Cyclic combinational circuits: Analysis for synthesis," in *Int'l Workshop Logic and Synthesis*, pp. 105–112, 2003.

[28] V. Agarwal, N. Kankani, R. Rao, S. Bhardwaj, and J. Wang, "An efficient combinationality check technique for the synthesis of cyclic combinational circuits," in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pp. 212–215, 2005.

[29] S. Thijssen, S. K. Jha, and R. Ewetz, "Equivalence checking for flow-based computing," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pp. 656–663, IEEE, 2022.

[30] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *Proceedings of the 38th annual Design Automation Conference*, pp. 530–535, 2001.

[31] Y. S. Mahajan, Z. Fu, and S. Malik, "Zchaff2004: An efficient sat solver," in *Theory and Applications of Satisfiability Testing: 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers 7*, pp. 360–375, Springer, 2005.

[32] G. Audemard and L. Simon, "On the glucose sat solver," *International Journal on Artificial Intelligence Tools*, vol. 27, no. 01, p. 1840001, 2018.

[33] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*, pp. 24–40, Springer, 2010.

[34] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, pp. 151–158, 1971.

[35] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking.," *Handbook of satisfiability*, vol. 185, no. 99, pp. 457–481, 2009.

[36] S. Yang, "Logic synthesis and optimization benchmarks user guide: Version 3.0," tech. rep., MCNC Technical Report, Jan. 1991.