In-Memory Execution of Compute Kernels using Flow-based Memristive Crossbar Computing

Dwaipayan Chakraborty Sunny Raj
Julio Cesar Gutierrez Troyle Thomas Sumit Kumar Jha
Computer Science Department
University of Central Florida
4000 Central Florida Blvd
Orlando, Florida, 32816
Email: {jha}@cs.ucf.edu

Abstract—**Rebooting computing using in-memory architectures relies on the ability of emerging devices to execute a legacy software stack. In this paper, we present our approach of executing compute kernels written in a subset of the C programming language using flow-based computing on nanoscale memristor crossbars. Our approach also tests the correctness of the design using the parallel Xyces electronic simulation software. We demonstrate the potential of our approach by designing and testing a compute kernel for edge detection in images.**

I. Introduction

The end of Dennard scaling [1], [2] has forced the society to aggressively explore opportunities for computing using emerging devices such as memristors and alternative paradigms such as stochastic computing. However, a sustainable approach to rebooting computing requires that we should be conscious of the socio-economic cost involved in exposing new hardware-software interfaces to programmers and application developers. We recall that the relatively simple Y2K bug cost the United States more than $200 billion [3] and the cost of adopting a new computer architecture could be too prohibitive. Hence, rebooting computing in the post-Moore era should allow application developers to program such computing systems using familiar programming methodologies.

Memristor crossbars provide an interesting fabric to perform in-memory computing as data can be stored for long periods of time without significant sustained use of energy. Flow-based crossbar computing [4]–[6] allows data to be loaded onto a memristor crossbar in a well-designed structural pattern such that the flow of current through the crossbar can be used to perform Boolean computations. Such an in-memory memristive computing approach may be useful for applications where the same computation needs to be performed on evolving data sets. However, it is not straightforward for an application developer to map even the simplest programs onto such nanoscale memristor crossbars.

Flow-based computing using nanoscale memristor crossbars is particularly suited for single-instruction multiple-data (SIMD) parallelism. The application developer can be expected to write *kernels* in a subset of the C language. Such a *kernel* should be algorithmically mapped onto a memristor crossbar – thereby, relieving the programmer of the need to learn a new programming model. Several interesting applications in computer vision, linear algebra and machine learning can benefit from such a compilation of restricted C programs to nanoscale memristor crossbars.

In this paper, we make the following contributions:

(i) We present a methodology for transforming compute kernels written in a subset of the C programming language into flow-based memristive crossbar computing designs.
(ii) We demonstrate our approach on a simplified edge detection example from image processing that admits SIMD parallelism. The example illustrates the potential of the proposed approach to exploit device-level parallelism.

II. Related Work

The characteristic pinched-hysteresis curve of memristors has been experimentally reported at least since the early 20th century [7]. In 1971, Leon Chua introduced memristor as a fourth fundamental electrical element based entirely on theoretical symmetry arguments [8]. The demise of Dennard scaling and the predicted demise of Moore's law for CMOS devices has led to an aggressive investigation into the design of nanoscale memristors [9]. Such nanoscale memristor devices are being investigated for the design of emerging computer architectures [5], [10], [11].

A memristor serves as a nanoscale resistive switch with non-volatile memory. Highly dense two-dimensional arrays of nanowires with nanoscale memristor switches located at the junctions of the nanowires serve as compact nanoscale memory systems with efficient energy consumption [12]. This crossbar architecture permits a large amount of data to be packaged into a small space. In this paper, we exploit such nanoscale memristor crossbars for implementing our in-memory computing infrastructure.

Memristors have been used to implement neuromporphic computing architectures with great success. General purpose computing using memristor crossbar arrays has been investigated. Recently, Programmable Logic-in-Memory (PLiM) based on a ReRAM crossbar array has been proposed to accelerate general-purpose computing [13]. The PLiM methodology exploits a variant of the native majority function as a basic instruction on top of a ReRAM crossbar. The PLiM compiler [14] transforms general-purpose code into a sequence of majority operations. Recently, Bhattacharjee et al. have carefully extended the PLiM approach to permit parallel execution of each word of a VLIW instruction set [15].

III. Background

Memristors have been used to design circuits that can evaluate arbitrary Boolean formula using a small family of simple logical operations [16]–[20], such as material implication and the universal NAND gate derived from the IMPLY logic. Nanoscale memristor circuit designs obtained by repeatedly applying such elementary operations have been used to automatically synthesize the ISCAS89 benchmark using Boolean Decision Diagrams and And-Inverter Graphs [10], [21]. However, these approaches require repeated switching of nanoscale memristors and often take more energy than approaches based on flow-based crossbar computing [4]–[6].

Flow-based crossbar computing exploits the fact that data stored in a memristor crossbar can cause the corresponding memristor to switch on (off) and connect (disconnect) the corresponding row and column nanowires. If the pattern in which data is loaded onto the crossbar is carefully designed, a flow of current from one nanowire to another is feasible if and only if a given Boolean formula evaluates to true on the data loaded in the crossbar. Our flow-based computing approach has led to the algorithmic design of a compact, fast, and energy-efficient one-bit adder [4].

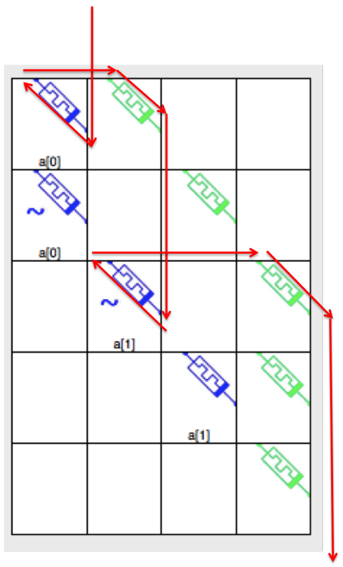

Fig. 1. The crossbar shows the second least significant bit of the increment operation. As a flow is injected in the leftmost nanowire, it reaches the rightmost nanowire if a[0]=1 and a[1]=0 i.e. incrementing a bit-vector ending in 01 produces a bit vector with the second least significant bit 1. Turned on memristors cause flows to jump from one nanowire to another as shown using the red arrow. The memristors not shown in the figure are all turned off. The memristors shown in green are turned on. The memristors shown in blue are labeled with the Boolean variable which should be loaded onto that memristor. The blue $\sim$ symbol denotes that the negation of the specified Boolean variable should be stored in the memristor.

In Figures 1 and 2, we illustrate the idea of flow-based computing with a small example. The crossbar computes the second least significant bit of incrementing a bit-vector by 1. The memristors that must remain turned-off at all times are not shown in the figure while the memristors that should always be turned-on are shown in green. The blue memristors capture the pattern of the data to be loaded on the crossbar and represent the (negated) Boolean variables that should be stored on these memristors. Each memristor stores a single Boolean value.

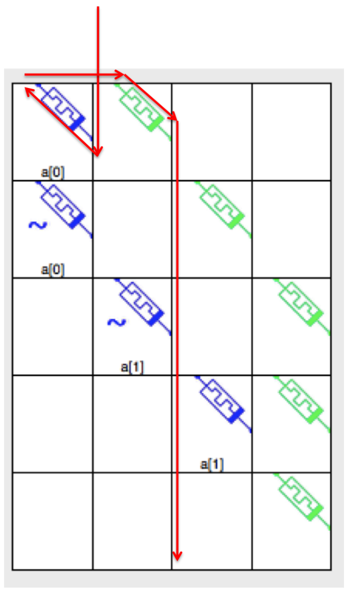

Fig. 2. The crossbar design corresponds to the second least significant bit of the increment operation. As a flow is injected in the leftmost nanowire, it doe not reach the rightmost nanowire if a[0]=1 and a[1]=1 i.e. incrementing a bit-vector ending in 11 produces a bit vector with the second least significant bit 0.

Figure 1 shows the flow of current using red arrows and indicates how incrementing a bit-vector ending in 10 (i.e. a[0] = 1 and $\sim$a[1]=1) leads to a flow on the rightmost nanowire. Figure 2 illustrates the scenario where the flow of current is blocked by turned-off memristors and does not reach the rightmost nanowire – thereby producing a logical 0. This shows that incrementing a bit-vector ending in 11 (i.e. a[0] = 1 and a[1]=1) does not turn on enough memristors to cause the flow to reach the rightmost nanowires and yields the correct logical value of 0 for the second least significant bit.

In our earlier work, we have devised scalable algorithms [5] for designing nanoscale memristor crossbars $\mathcal{C}(\phi)$ from Boolean logic specification $\phi(x_1, x_2, \cdots, x_k)$ that obey this property: the current flow introduced at a specific nanowire of the crossbar $\mathcal{C}(\phi)$ reaches another specific nanowire if and only if the Boolean formula $\phi(x_1, x_2, \cdots, x_k)$ evaluates to true for the inputs $x_1, x_2, \cdots, x_k$. In this paper, we leverage these automated synthesis algorithms to design nanoscale memristor crossbars for in-memory computing from compute kernels written using the C programming language.

IV. Approach

Our approach uses the LLVM compilation framework to transform compute kernel code in the C programming language into the LLVM intermediate representation. The LLVM IR is then transformed into a Boolean Decision Diagram (BDD) and the decision diagram is then mapped onto a nanoscale memristor crossbar. We exploit the parallel Xyces electronic simulation software to test the correctness of our designed nanoscale memristor crossbars.

A. Compute Kernel

The compute kernel has a structure representing the inputs that it needs to use. Each input can be an integer whose bit width is specified during compilation. In our example code, the kernel for edge detection has two inputs: p1 representing one pixel value and p2 representing a pixel value next to p1.

The kernel also has an output structure that has at least one output variable. Each output variable is an integer whose bit width is again specified during compilation. In our example, the output structure has a single integer output o.

```
// Edge detection kernel
struct Input { int p1; int p2;} input;
struct Output{int o; } output;

void compute()
{
  output.o = input.p2 - input.p1;
}
```

.

The core of the kernel is the compute() function that uses the syntax of the C language to define how the input variables must be manipulated to produce the output variable. The C code inside the kernel should not have loops that cannot be statically unrolled.

B. From C Kernel to Intermediate Representation

The C kernel code is compiled using the LLVM compiler to an intermediate representation. The structure of the kernel definition allows us to readily obtain the core computations being performed by the kernel code.

```

define void @compute() #0 {
  %1 = load i32, i32* getelementptr inbounds (%
     struct.Input, %struct.Input* @input, i64 0, i32
     0), align 4, !tbaa !1
  %2 = load i32, i32* getelementptr inbounds (%
     struct.Input, %struct.Input* @input, i64 0, i32
     1), align 4, !tbaa !6
  %3 = sub nsw i32 %1, %2
  store i32 %3, i32* getelementptr inbounds (%struct
     .Output, %struct.Output* @output, i64 0, i32 0),
      align 4, !tbaa !7
  ret void
}
```

.

The compiled intermediate representation for our running example at the bottom of the left column. The LLVM intermediate representation provides a simple linear list of arithmetic and logical operations that can then be transformed to Boolean Decision Diagrams.

C. Intermediate Representation to Boolean Decision Diagram

Boolean Decision Diagram packages like BuDDy provide robust algorithms for mapping logical and linear arithmetic operations into Boolean Decision Diagrams [22], [23]. The LLVM intermediate representation obtained from our computational kernel makes direct references to arithmetic and logical computations on its operands and enables a one-one mapping to Boolean Decision Diagram operations.

In our running example, the LLVM intermediate representation states " %3 = sub nw i32 %1 %2 " indicating that a subtraction operation should be performed using the operands in the first two positions on the input structure and the result returned as the first element of the output structure. This is readily achieved in our approach using the bit-vector subtraction operation in the BuDDy package.

Nonlinear arithmetic operations like multiplication lead to an explosion in the size of the BDD as the number of bits in the input increases. Hence, the approach based on BDDs is not efficiently applicable to such nonlinear computations.

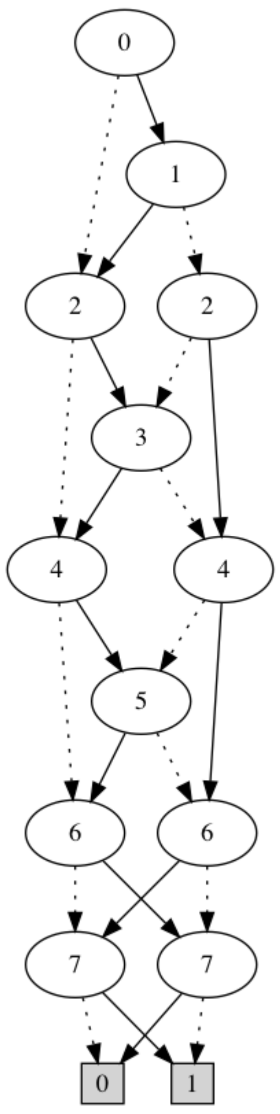

Fig. 3. Boolean Decision Diagram representing the most significant bit of the subtraction of two 4-bit integers. Alternate layers of the BDD interleave the Boolean variables corresponding to the two operands for efficiency.

D. Boolean Decision Diagram to Memristor Crossbar

The flow of information through a Boolean Decision Diagram can be mapped to the flow of current through a nanoscale memristor crossbar [5]. The structure of the BDD is used to design the pattern in which data should be stored onto the crossbar. A flow of current may then be injected onto one nanowire and it will reach another specified nanowire if and only if the corresponding bit of the output should be true.

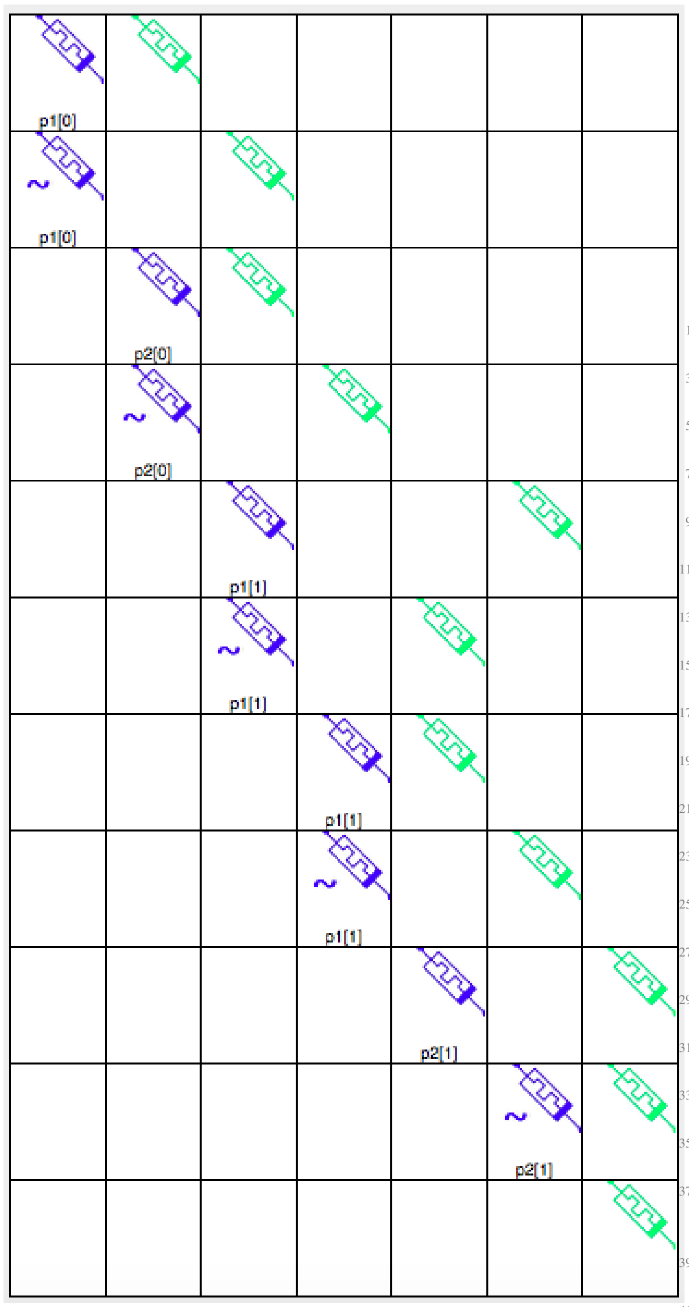

Fig. 4. Crossbar design representing the second least-significant bit of the subtraction of two 4-bit integers. The memristors not shown in the figure are all turned off. The memristors shown in green are turned on. The memristors shown in blue are labeled with the Boolean variable which should be loaded onto that memristor. The blue ∼ symbol denotes that the negation of the specified Boolean variable should be stored in the memristor.

Using our running kernel example, Figure 4 shows the design of the nanoscale memristor crossbar that can be used to compute the second least significant bit of the nanoscale memristor crossbar. The memristors colored in green are always turned on regardless of the value of the input variables. For the sake of clarity, memristors that are perpetually turned off are not drawn in the figure. The memristors colored in blue are labeled either with a variable or its negation – they are turned on or off depending on the values of the input variables on which computation is being performed.

E. Testing of Crossbars using Xyces

The crossbar designed using the Boolean Decision Diagram representation of the kernel code is tested for correctness using the parallel Xyces electronic simulation software. The listing below illustrates the simulation code when p1=1011 and p2=1000.

```

**** OUTPUT VARIABLE: o BIT: 0 ****
*
VCC c0 0 2V
*
*Resistors
Rr0 r0 0 10K
Rr1 r1 0 10K
Rr2 r2 0 10K
Rr3 r3 0 10K
Rr4 r4 0 10K
Rc0 c0 0 10K
Rc1 c1 0 10K
Rc2 c2 0 10K
Rc3 c3 0 10K
R_OUT c3 0 1
*
*turned-on memristors
YMEMRISTOR MR43 r4 c3 MEMModel XO=1.0
YMEMRISTOR MR01 r0 c1 MEMModel XO=1.0
YMEMRISTOR MR12 r1 c2 MEMModel XO=1.0
YMEMRISTOR MR23 r2 c3 MEMModel XO=1.0
YMEMRISTOR MR33 r3 c3 MEMModel XO=1.0
*
*Memristors dependent on variables
YMEMRISTOR MR00_p10T r0 c0 MEMModel XO=1.0
YMEMRISTOR MR10_p10Fn r1 c0 MEMModel XO=1n
YMEMRISTOR MR21_p20Tn r2 c1 MEMModel XO=1.0
YMEMRISTOR MR32_p20F r3 c2 MEMModel XO=1n
*
*OFF
*
*Transient Anaylsis
.TRAN 250ms 600ms 0ms
*
*Output
.PRINT TRAN FORMAT=NOINDEX PRECISION=1 DELIMITER=TAB
     TIMESCALEFACTOR=1000
+V(c3,0)
+I(R_OUT)
*
*
.MODEL MEMModel memristor level=3 a1=0.2 a2=0.05 b
    =0.05 vp=2.2 vn=2.2
+ap=1.9e9 an=1.9e9 xp=0.675 xn=0.675 alphap=0.01
    alphan=0.01 eta=1
*
.END
```

The expected output for this pair of inputs is 1101 and the currents produced by the Xyces simulation are 1.9e-03 (flow), 1.8e-11 (no flow), 4.3e-4 (flow) and 1.9e-4 (flow) amperes. Hence, the design works correctly in this case and produces the expected output of 1101.

The design has been shown to be correct on all possible inputs. We demonstrate the quantitative nature of these results using two more representative examples. We test the design with the inputs p1=0010 and p2=1110. The flow of current observed by the Xyces simulation package are 9.0e-12, 2.6e-11, 4.7e-4, and 2.5e-4 amperes; these values correspond to the correct output of 1100. Similar results are obtained when the crossbar design of our running kernel example is tested using the inputs p1=0101 and p2=1111. The expected correct output is 1010 and it is in conformance with our observed flow values using the Xyces simulator: 9.0e-12, 7.3e-4, 8.2e-12, and 1.9e-04 amperes.

Validation using the Xyces simulator is meant to isolate errors in implementations of our design flow from the kernel code to the crossbar. It also enables us to theoretically investigate the number of bits in a computation that can be implemented on a memristor crossbar with a given ratio of the high resistance state (HRS) to the low resistance state (LRS). State-of-the-art memristors with a HRS/LRS ratio of 100,000 have been reported; hence, it should be possible to implement computations with 10,000 bits on a single crossbar. Further, millions of SIMD instructions with 10,000 or fewer bits can be executed in parallel on different data sets on the same nanoscale memristor crossbar. The Xyces simulation environment is parallelized and allows rapid theoretical prototyping of our designs.

V. Illustrative Example

We wrote a compute kernel in our framework to implement a simple edge detection framework. The inputs to our kernel include the RGB values of two pixels and the output is 4 less than the different of the sum of the RGB values of the two pixels. Intuitively, if this output is less than 0, there is no edge at the location of this pixel.

```
// RGB edge detection kernel code
struct Input { int r1; int g1; int b1; int r2; int
    g2; int b2; } input;
struct Output{int o; } output;

void compute()
{
  int avg1, avg2, diff;
  avg1 = input.r1 + input.g1 + input.b1;
  avg2 = input.r2 + input.g2 + input.b2;
  diff = avg2-avg1;
  output.o = diff -4;
}
```

.

Using our approach, we transformed the above code into its equivalent LLVM intermediate representation:

```
  %7 = sub i32 -4, %1
  %8 = sub i32 %7, %2
  %9 = sub i32 %8, %3
  %10 = add i32 %9, %4
  %11 = add i32 %10, %5
  %12 = add i32 %11, %6
```

.

We then use the BuDDy package to convert the LLVM intermediate representation into its Boolean Decision Diagram representation using 4-bit integers. Using the Xyces simulation software, we tested the generated crossbar on the following inputs: b1 = 0001, b2 = 0101, g1 = 1000, g2 = 1001, r1 = 1001, and r2 = 1101. The expected correct output is 0101 and the current flows observed by Xyces was in agreement: 3.3e-04 (flow), 2.1e-12 (no flow), 3.3e-06 (flow), 1.9e-14 (no flow) amperes. For this program, the sizes of the crossbar for the first four bits are 21×12, 55×29, 103×53, and 171×87. All pixels of an image can be processed in parallel on such crossbars and the edge detection operation can be performed in constant time (independent of the size of the image).

VI. Conclusions and Future Work

Compute kernels rich in logical and linear arithmetic computations can be automatically compiled into flow-based crossbar computing designs. Existing robust compilation frameworks like LLVM and well-studied Boolean Decision Diagram packages like BuDDy can be used to implement such a transformation of well-designed compute kernels into in-memory computing accelerators without any involvement of the application developer.

Several interesting theoretical and practical directions for future research into automated synthesis of flow-based computing systems remain open. First, our compilation prototype is not robust enough to be deployed in a real-world setting due to its poor support for compilation errors and insufficient testing on a variety of examples. Second, nonlinear arithmetic such as multiplication and division are the bottlenecks in many software systems and the search of compact memristor crossbars for executing such nonlinear operations would be of tremendous value to the proposed in-memory execution of compute kernels. Reduced Ordered Boolean Decision Diagrams (ROBDDs) can not succinctly represent the multiplication operation; hence, a richer data structure capable of compactly representing nonlinear operations is needed. Third, the current approach does not deal with important practical fault-tolerance issues. It should be feasible to extend this compilation approach to fault tolerant flow-based computing crossbars [24] such that the nodes of the Boolean Decision Diagram being mapped to the crossbar avoid memristors that are known to be faulty.

VII. Acknowledgement

The authors would like to thank the US Air Force for support provided through the AFOSR Young Investigator Award to Sumit Jha. The authors acknowledge support from the National Science Foundation Software & Hardware Foundations #1438989 and Exploiting Parallelism & Scalability #1422257 projects. This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-16-1-0255. The authors are grateful to Michael Claudio and Austin Miller for their help in implementing a web front-end for the prototype of this tool.

References

[1] D. J. Frank, R. H. Dennard, E. Nowak, P. M. Solomon, Y. Taur, and H.-S. P. Wong, "Device scaling limits of si mosfets and their application dependencies," *Proceedings of the IEEE*, vol. 89, no. 3, pp. 259–288, 2001.

[2] M. D. Hill and M. R. Marty, "Retrospective on amdahl's law in the multicore era," *Computer*, vol. 50, no. 6, pp. 12–14, 2017.

[3] R. Mitchell, "Y2k: The good, the bad and the crazy," *ComputerWorld (December 2009)*, 2009.

[4] Z. Alamgir, K. Beckmann, N. Cady, A. Velasquez, and S. K. Jha, "Flow-based computing on nanoscale crossbars: Design and implementation of full adders," in *Circuits and Systems (ISCAS), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 1870–1873.

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

[6] ——, "Design of compact memristive in-memory computing systems using model counting," in *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 2655–2658.

[7] G. Gandhi, V. Aggarwal, and L. Chua, "Coherer is the elusive memristor," in *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 2245–2248.

[8] L. Chua, "Memristor-the missing circuit element," *IEEE Transactions on circuit theory*, vol. 18, no. 5, pp. 507–519, 1971.

[9] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *nature*, vol. 453, no. 7191, pp. 80–83, 2008.

[10] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Fast logic synthesis for RRAM-based in-memory computing using majority-inverter graphs," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. EDA Consortium, 2016, pp. 948–953.

[11] 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, 2014.

[12] W. Kim, A. Chattopadhyay, A. Siemon, E. Linn, R. Waser, and V. Rana, "Multistate memristive tantalum oxide devices for ternary arithmetic," *Scientific Reports*, vol. 6, 2016.

[13] P.-E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, "The programmable logic-in-memory (plim) computer," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. EDA Consortium, 2016, pp. 427–432.

[14] M. Soeken, S. Shirinzadeh, P.-E. Gaillardon, L. G. Amarú, R. Drechsler, and G. De Micheli, "An MIG-based compiler for programmable logic-in-memory architectures," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 2016, pp. 1–6.

[15] D. Bhattacharjee, R. Devadoss, and A. Chattopadhyay, "ReVAMP: ReRAM based VLIW architecture for in-memory computing," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017*, 2017.

[16] E. Gale, B. de Lacy Costello, and A. Adamatzky, *Boolean Logic Gates from a Single Memristor via Low-Level Sequential Logic*. Springer, 2013, pp. 79–89.

[17] S. Kvatinsky, A. Kolodny, U. C. Weiser, and E. G. Friedman, "Memristor-based IMPLY logic design procedure," in *Computer Design (ICCD), 2011 IEEE 29th International Conference on*. IEEE, 2011, pp. 142–147.

[18] E. Lehtonen and M. Laiho, "Stateful implication logic with memristors," in *Proceedings of the 2009 IEEE/ACM International Symposium on Nanoscale Architectures*. IEEE Computer Society, 2009, pp. 33–36.

[19] X. Sun, G. Li, L. Ding, N. Yang, and W. Zhang, "Unipolar memristors enable "stateful" logic operations via material implication," *Applied Physics Letters*, vol. 99, no. 7, p. 072101, 2011.

[20] C. Xu, X. Dong, N. P. Jouppi, and Y. Xie, "Design implications of memristor-based RRAM cross-point structures," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 2011, pp. 1–6.

[21] S. Chakraborti, P. V. Chowdhary, K. Datta, and I. Sengupta, "BDD based synthesis of boolean functions using memristors," in *Design & Test Symposium (IDT), 2014 9th International*. IEEE, 2014, pp. 136–141.

[22] S. B. Akers, "Binary decision diagrams," *IEEE Trans. Computers*, vol. 27, no. 6, pp. 509–516, 1978.

[23] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys (CSUR)*, vol. 24, no. 3, pp. 293–318, 1992.

[24] A. Velasquez and S. K. Jha, "Fault-tolerant in-memory crossbar computing using quantified constraint solving," in *Computer Design (ICCD), 2015 33rd IEEE International Conference on*. IEEE, 2015, pp. 101–108.